

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

І. В. ФЕДОРІН

**МЕТОДИ ТА
ТЕХНОЛОГІЇ
ОБЧИСЛЮВАЛЬНОГО
ІНТЕЛЕКТУ**

Навчальний посібник

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів другого (магістерського) рівня
вищої освіти за освітньою програмою «Комп'ютерні технології в біології
та медицині» спеціальності 122 «Комп'ютерні науки»*

Київ
КПІ ім. Ігоря Сікорського
2022

Рецензент *Олійник А.С.*, д.фіз.-мат.н., доцент, професор кафедри алгебри і комп'ютерної математики Київського національного університету імені Тараса Шевченка

Сахненко Н.К., д.фіз.-мат.н., доцент, доцент кафедри математичного моделювання та аналізу даних, Навчально-науковий фізико-технічний інститут, Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

Відповідальний редактор *Носовець О. К.*, к.т.н., доцент кафедри біомедичної кібернетики, Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

*Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 2 від 30.09.2022 р.)*

*за поданням Вченої ради факультету біомедичної інженерії
(протокол №1 від 01.09.2022 р.)*

Електронне мережне навчальне видання

Федорін Ілля Валерійович, канд. фіз.-мат. наук

МЕТОДИ ТА ТЕХНОЛОГІЇ ОБЧИСЛЮВАЛЬНОГО ІНТЕЛЕКТУ

НАВЧАЛЬНИЙ ПОСІБНИК

Методи та технології обчислювального інтелекту: Навчальний посібник [Електронний ресурс] : навч. посіб. для студ. спеціальності 122 «Комп'ютерні науки» / І. В. Федорін; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 8,7 Мбайт). – Київ: КПІ ім. Ігоря Сікорського, 2022. – 314 с.

Навчальний посібник призначено для оволодіння теоретичними основами методів та алгоритмів обчислювального інтелекту. Надані загальні теоретичні відомості з теорії обчислювального інтелекту та машинного навчання та практичні рекомендації щодо розробки та застосування відповідних алгоритмів та систем. У навчальному посібнику врахований сучасний стан розвитку алгоритмів та мереж глибокого навчання.

© І. В. Федорін, 2022

© КПІ ім. Ігоря Сікорського, 2022

ЗМІСТ

Вступ.....	8
Тема №1: Нейрон та нейронна мережа. Математична модель штучного нейрона	10
Біологічний нейрон. Нервова система людини.....	10
Структура нейрона.	11
Взаємодія між нейронами.	14
Структура простої рефлекторної біологічної нейронної мережі.	15
Математична модель нейрона.....	16
Докладніше, що таке нейрон зсуву (використовується поняття функції активації, яке більш детально надано наприкінці розділу).	19
Функції активації. Види функції активації.....	21
Як вибрати функцію активації?	30
Нейронне представлення логічних вентилів AND, OR, NOT, XOR та XNOR (алгоритм персептрона, часте питання на співбесідах)	30
Тема №2: Навчання нейронної мережі. Функція втрат задачі регресії. Глибока нейронна мережа та їх різновиди.	42
Етапи життєвого циклу штучної нейронної мережі (ШНМ).....	42
Скорочена модель побудови нейронної мережі	45
Навчання нейронних мереж	47
Загальні відомі правила навчання ШНМ.....	50
Типи завдань машинного навчання та ШНМ.....	53
Метрики якості. Функції втрат у завдання регресії.....	54
Глибокі нейронні мережі.....	66
Прокляття розмірності. Недонавчання та перенавчання.	69

Як зробити, щоб ШНМ давала правильні відповіді? Градієнтний спуск, зворотне поширення помилки	72
Методи навчання	89
Що таке гіперпараметри?	89
Що таке збіжність?	90
Різноманітність нейронних мереж.	91
Тема №3: Задача класифікації. Функції втрат задач класифікації.	106
Завдання класифікації	106
Функції втрат задач класифікації	111
Тема №4: Регуляризація функції втрат. Нормалізація даних. Batch - нормалізація	129
Регуляризація	129
Нормалізація даних, передобробка.	140
Батч-нормалізація (batch normalization)	147
Інші способи боротьби з перенавчанням	148
Ініціалізація мережі	150
Тема №5: Метод максимальної правдоподібності	153
Загальні визначення. Постановка завдання	153
Обчислення оцінок максимальної правдоподібності	156
Корисні зауваження	159
Тема №6: Методи оптимізації	167
Градієнтний спуск.	168
Стохастичний градієнтний спуск (SGD або CGC)	177
Модифікації градієнтного спуску	178
Інерційні чи прискорені градієнтні методи	179

Proximal методи.....	185
Алгоритми градієнтної оптимізації.....	186
Інші методи.....	192
Візуалізація алгоритмів оптимізації.....	192
Який оптимізатор використати?.....	193
Використання генетичного алгоритму для оптимізації рекурентних нейронних мереж (базовий огляд методу на прикладі).	194
Генетичний алгоритм.....	194
Реалізація генетичного алгоритму на прикладі RNN моделі	196
Оптимізація повнозв'язкової нейронної мережі (MLP) генетичним алгоритмом.....	201
Тема №7: Згортка. Згортковий шар та згорткова нейронна мережа (CNN).	208
Вступ. Загальні зауваження.	208
Загальна будова згорткової нейронної мережі (CNN).	212
Згортковий шар.....	213
Об'єднання (Pooling)	214
Повнозв'язковий шар	214
Рецептивне поле	214
Вага	214
Як працює CNN – математичний апарат на прикладі.....	215
Згортка.....	215
Striding	220
Padding.....	221
Pooling. Просторове об'єднання.....	223

Параметри в згортковій нейронній мережі.....	224
Рецептивні поля.....	228
Нелінійність (ReLU).....	231
Повністю пов'язаний шар	232
Проблема adversarial attacks	233
Тема №8: Архітектури згорткових нейронних мереж.	236
LeNet5	237
Dan Ciresan Net	238
AlexNet	238
Overfeat	242
VGG	242
Network-in-network	246
GoogLeNet та Inception V1	246
Inception V2 та V3	248
ResNet	251
Inception V4.....	257
SqueezeNet.....	257
ENet.....	258
Xception	258
MobileNets	259
FractalNet	260
U-Net: нейромережа для сегментації зображень	260
NBDT: нейромережа на основі вирішальних дерев	264
Підсумок. Використовуйте згорткові мережі!	265
Тема №9: Рекурентні нейронні мережі.	267

Що таке рекурентні нейронні мережі?.....	267
Навчання RNN.....	270
Модифікації RNN.....	270
LSTM –мережа з довгою короткостроковою пам'яттю.....	271
GRU – керований рекурентний нейрон	280
Тема №10: Генеративно-змагальна нейромережа (GAN).....	282
Загальні положення.....	282
Дискримінатор та генератор.	283
Як працюють GAN	283
Інтуїтивний процес тренування.....	286
Поліпшення навчання GAN	287
Автокодери та VAE.....	288
Поради щодо навчання GAN	288
Приклад коду	289
Деякі різновиди архітектур GAN.....	293
Conditional Generative Adversarial Nets, CGAN.....	293
DCGAN (Deep Convolutional Generative Adversarial Nets).....	294
StackGAN	296
LAPGAN (Laplacian Pyramid of Adversarial Networks).....	298
Список використаних джерел	301

ВСТУП

Основою метою вивчення навчальної дисципліни «Методи та технології обчислювального інтелекту» є формування у студентів розуміння принципів, вмінь та практичних навичок, які дозволять спроектувати медичну систему штучного інтелекту, яка виконує наступні функції:

- 1) автоматично обробляє великий обсяг персоналізованої інформації з метою подальшого аналізу
- 2) за результатами автоматичної обробки даних надає персоналізовані рекомендації
- 3) за результатами автоматичної обробки з певною вірогідністю надає скринінговий результат дослідження
- 4) використовує нові дані, для автоматичного донавчання системи
- 5) являє собою зв'язковою ланкою системи користувач-медичний працівник

Згідно з вимогами програми навчальної дисципліни студенти після засвоєння кредитного модуля мають продемонструвати такі результати навчання:

Знання:

- основних концепцій та технологій обчислювального інтелекту
- знання критеріїв якості та ефективності використання методів обчислювального, штучного інтелекту та машинного навчання
- основних алгоритмів моделювання складних біомедичних систем та процесів
- основ оптимізації роботи алгоритмів та методів
- типів та особливостей архітектур нейронних мереж

Уміння:

- використовувати методи штучного та обчислювального інтелекту до проектування, розробки та тестування програмних продуктів біомедичного спрямування

- забезпечити пошук, обробку та зберігання біомедичної інформації та баз даних у бажаному форматі, використовуючи мережеві системи
- ідентифікувати об'єкт дослідження, порядок зберігання та обробки даних методами штучного інтелекту для вирішення задач біомедичного спрямування
- моделювати біомедичні системи методами обчислювального інтелекту та машинного навчання
- визначати рівень достовірності та ефективності роботи розробленого програмного продукту на основі методів обчислювального інтелекту та машинного навчання
- визначати критерії якості та основні характеристики об'єкту дослідження та програмного продукту біомедичного спрямування розробленого з використанням методів обчислювального та штучного інтелекту.
- ідентифікувати алгоритми, моделі та методи штучного інтелекту які підходять для вирішення тієї чи іншої задачі біомедичного спрямування
- забезпечити декомпозицію задачі біомедичного спрямування на складові елементи з метою застосування відомих методів і технологій для її вирішення
- оптимізувати роботу обраних алгоритмів та методів штучного інтелекту

У даному навчальному посібнику надані теоретичні основи систем штучного інтелекту, основ машинного навчання, глибокого навчання, базових та сучасних архітектур нейронних мереж. Оволодіння теоретичним матеріалом, який наведено у навчальному посібнику, дозволить отримати знання, необхідні для самостійної розробки та впровадження алгоритмів машинного навчання та нейронних мереж у задачах галузі охорони здоров'я.

ТЕМА №1: НЕЙРОН ТА НЕЙРОННА МЕРЕЖА. МАТЕМАТИЧНА МОДЕЛЬ ШТУЧНОГО НЕЙРОНА

Біологічний нейрон. Нервова система людини.

Нервова система координує діяльність клітин, тканин та органів нашого тіла. Вона регулює функції організму та його взаємодію з навколишнім середовищем, забезпечує можливості реалізації психічних процесів, що лежать в основі механізмів мови та мислення, запам'ятовування та навчання. Крім того, у людини нервова система становить матеріальну основу її психічної діяльності.

Нервова система є складним комплексом високоспеціалізованих клітин, що передають імпульси від однієї частини тіла до іншої, в результаті організм отримує можливість реагувати як єдине ціле на зміни факторів зовнішнього або внутрішнього середовища [1].

Нервова система поділяється на дві частини: центральну та периферичну.

До складу *центральної нервової системи* входять головний та спинний мозок. До складу *периферичної нервової системи* – нерви, нервові вузли та нервові закінчення.

Спинний мозок являє собою довгастий, циліндричний тяж довжиною до 45 см і масою 34-38 г, що знаходиться в хребетному стовпі. Його верхня межа розташована біля основи черепа (верхні відділи переходять у головний мозок), а нижня – у I-II поперекових хребців. Від спинного мозку симетрично відходять корінці спинномозкових нервів. У ньому знаходяться центри деяких простих рефлексів, наприклад рефлексів, що забезпечують рух діафрагми, дихальних м'язів. Спинний мозок виконує дві функції: *рефлекторну та провідну*, та під контролем головного мозку регулює роботу внутрішніх органів (серця, нирок, органів травлення).

Периферична нервова система функціонально та структурно поділяється на соматичну нервову систему та вегетативну нервову систему. Соматична нервова система відповідає за координацію рухів тіла, і навіть отримання зовнішніх стимулів. Це система, яка регулює свідомо контрольовану діяльність.

Вегетативна нервова система, у свою чергу, ділиться на симпатичну нервову систему, парасимпатичну нервову систему та ентеральну нервову систему. Симпатична нервова система відповідає за реагування на небезпеку або стрес, що насувається, і разом з іншими фізіологічними змінами відповідає за збільшення частоти пульсу і кров'яного тиску, а також при появі почуття хвилювання сприяє підвищенню рівня адреналіну. Парасимпатична нервова система, навпаки, стає помітною, коли людина відпочиває і відчувається спокійно, вона відповідає за такі речі, як звуження зіниць, уповільнення серцебиття, розширення кровоносних судин та стимуляцію роботи травної та сечостатевої систем. Роль ентеральної нервової системи полягає в управлінні всіма аспектами травлення, від стравоходу до шлунку, тонкого кишечника та прямої кишки.

Основним структурним і функціональним елементом нервової системи є нервові клітини - нейрони! Сукупність нейронів та міжклітинної речовини утворює нервову тканину [1].

Для довідки.

- нервова система складається із 10...100 мільярдів нервових клітин;
- мозок споживає близько 10 Вт енергії (еквівалентно потужності нічної лампи) та за 1 хв. через нього протікає 740-750 мл крові;
- нервові клітини генерують приблизно до тисячі імпульсів за секунду.

Структура нейрона.

Елементом клітинної структури мозку є нервова клітина – нейрон. Нейрон у своїй будові має багато спільних рис з іншими клітинами біотканини: тіло нейрона оточене плазматичною мембраною, усередині якої

знаходиться цитоплазма, ядро та інші складові клітини. Однак нервова клітина істотно відрізняється від інших за своїм функціональним призначенням. Нейрон виконує прийом, елементарне перетворення та подальшу передачу інформації іншим нейронам. Інформація переноситься як імпульсів нервової активності, має електрохімічну природу.

Нейрони вкрай різноманітні за формою, яка залежить від їхнього місцезнаходження в нервовій системі та особливостей функціонування. На рисунку 1.1 наведено схему будови "типового" нейрона. Тіло клітини містить безліч відростків двох типів, що гілкуються. Відростки першого типу, які називають дендритами за їх схожість з кроною розлогого дерева, служать як входні канали для нервових імпульсів від інших нейронів. Ці імпульси надходять у сому або тіло клітини розміром від 3 до 100 мікрон, викликаючи її специфічне збудження, яке потім поширюється вивідним відростком другого типу – аксону. Довжина аксонів зазвичай помітно перевищує розміри дендритів, в окремих випадках досягаючи десятків сантиметрів і навіть метрів. Гігантський аксон кальмара має товщину близько міліметра, і саме спостереження за ним послужило дало змогу з'ясувати механізм передачі нервових імпульсів між нейронами.

Тіло нейрона, заповнене провідним іонним розчином, оточене мембраною товщиною близько 75 ангстрем, що має низьку провідність. Між внутрішньою поверхнею мембрани аксона та зовнішнім середовищем підтримується різниця електричних потенціалів. Це здійснюється за допомогою молекулярного механізму іонних насосів, що створюють різну концентрацію позитивних іонів K^+ та Na^+ усередині та поза клітиною. Проникність мембрани нейрона селективна для цих іонів. Усередині аксона клітини, що перебуває у стані спокою, активний транспорт іонів прагне підтримувати концентрацію іонів калію вищою, ніж іонів натрію, тоді як у рідині, що оточує аксон, вище виявляється концентрація іонів Na^+ . Пасивна дифузія більш рухливих іонів калію призводить до їхнього інтенсивного

виходу з клітини, що зумовлює її загальний негативний (щодо зовнішнього середовища) потенціал спокою, що становить близько – 65 мілівольт [1-5].

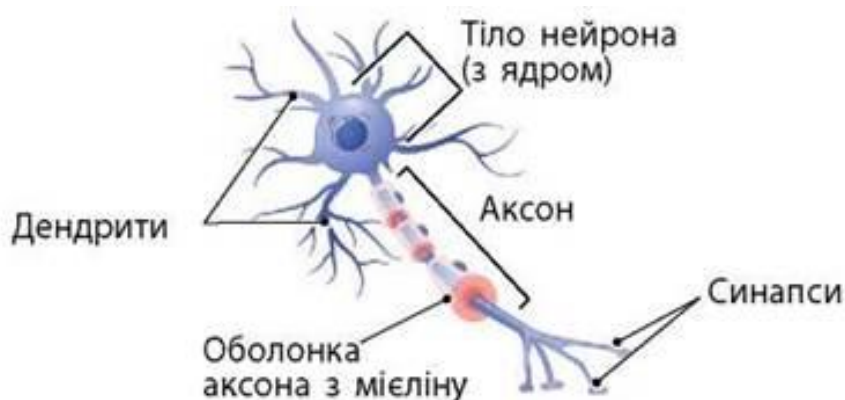


Рисунок 1.1 – Будова нейрона [1]

Під впливом стимулюючих сигналів з інших нейронів мембрана аксона динамічно змінює свою провідність. Це відбувається, коли сумарний внутрішній потенціал перевищує граничне значення масштабу -50 мВ. Мембрана на короткий час, що становить близько 2 мілісекунд, змінює свою полярність (деполяризується) і досягає потенціалу дії близько +40 мВ. На мікрорівні це пояснюється короткочасним підвищенням проникності мембрани для іонів Na^+ та активним надходженням їх до аксону. Надалі, у міру виходу іонів калію, позитивний заряд з внутрішньої сторони мембрани змінюється на негативний і настає так званий період рефрактерності, що триває близько 200 мс. Протягом цього часу нейрон є повністю пасивним, практично незмінно зберігаючи потенціал усередині аксона лише на рівні -70 мВ [4].

Імпульс деполяризації клітинної мембрани, що зветься «спайком», поширюється вздовж аксона практично без загасання, підтримуючись локальними іонними градієнтами. Швидкість переміщення спайка є відносно невисокою та становить від 100 до 1000 сантиметрів на секунду.

Збудження нейрона у вигляді спайка передається іншим нейронам, які таким чином поєднані в мережу, яка проводить нервові імпульси. Ділянки

мембрани на аксоні, де розміщуються області контакту аксона даного нейрона з дендритами та іншими нейронами, називаються *синапсами*. В області синапсу, що має складну будову, відбувається обмін інформацією про порушення між нейронами. Механізми синаптичної передачі досить складні та різноманітні. Вони можуть мати хімічну та електричну природу. У хімічному синапсі у передачі імпульсів беруть участь специфічні хімічні речовини – нейромедіатори, які викликають зміни проникності локальної ділянки мембрани. Залежно від типу медіатора, що виробляється, синапс може мати збуджуючу (ефективно проводить збудження) або гальмуючу дію. Зазвичай на всіх відростках одного нейрона виробляється один й той самий медіатор, і тому нейрон в цілому функціонально є гальмуючим або збуджуючим. Це важливе спостереження про наявність нейронів різних типів у наступних розділах істотно використовуватиметься при проектуванні штучних нейронних мереж [6].

Взаємодія між нейронами.

Взаємодіючи між собою за допомогою передачі через відростки збуджень нейрони формують нейронні мережі. Перехід від розгляду окремого нейрона до вивчення нейронних мереж є природним кроком у нейробіологічній ієрархії.

Загальна кількість нейронів у центральній нервовій системі людини досягає 10^{10} - 10^{11} , при цьому кожна нервова клітина пов'язана в середньому з 10^3 - 10^4 інших нейронів. Встановлено, що в головному мозку сукупність нейронів обсягом 1 мм^3 формує відносно незалежну локальну мережу, що несе певне функціональне навантаження.

Виділяють кілька (зазвичай три) основних типів нейронних мереж, що відрізняються структурою та призначенням. **Перший тип** становлять ієрархічні мережі, що часто зустрічаються в сенсорних та рухових системах. Інформація у таких мережах передається у процесі послідовного переходу від одного рівня ієрархії до іншого.

Нейрони утворюють два характерні типи сполук – *конвергентні*, коли велика кількість нейронів одного рівня контактує з меншим числом нейронів наступного рівня, і *дивергентні*, в яких контакти встановлюються з дедалі більшим числом клітин наступних верств ієрархії. Поєднання конвергентних та дивергентних сполук забезпечує багаторазове дублювання інформаційних шляхів, що є вирішальним фактором надійності нейронної мережі. При загибелі частини клітин, нейрони, що збереглися, виявляються в змозі підтримувати функціонування мережі.

До другого типу нейронних мереж відносяться локальні мережі, які формуються нейронами з обмеженими сферами впливу. Нейрони локальних мереж проводять переробку інформації у межах одного рівня ієрархії. При цьому функціонально локальна мережа є відносно ізольованою гальмуючою або збудливою структурою. Важливу роль також грають так звані дивергентні мережі з одним входом. Командний нейрон, що знаходиться в основі такої мережі може впливати відразу на безліч нейронів, і тому мережі з одним входом виступають узгоджувачим елементом у складному поєднанні нейромережових систем усіх типів.

Структура простої рефлекторної біологічної нейронної мережі.

Розглянемо схематично нейронну мережу, що формує простий рефлекторний ланцюг з передачею збудження від подразника до рухового м'яза (рисунок 1.2) [1].

Сигнал зовнішнього подразника сприймається сенсорними нейронами, пов'язаними з чутливими клітинами-рецепторами. Сенсорні нейрони формують перший (нижній) рівень ієрархії. Вироблені ними сигнали передаються нейронам локальної мережі, що містять безліч прямих та зворотних зв'язків із поєднанням дивергентних та конвергентних сполук. Характер перетвореного у локальних мережах сигналу визначає стан збудження моторних нейронів. Ці нейрони, що становлять верхній рівень

ієрархії, образно кажучи, "приймають рішення", яке виражається у впливі на клітини м'язової тканини за допомогою нервово-м'язових сполук.

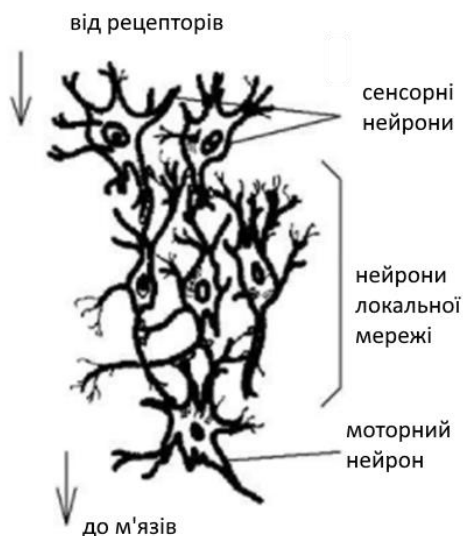


Рисунок 1.2 – Структура рефлекторної нейронної мережі

Математична модель нейрона.

Отже, основними дієвими елементами нервової системи є окремі клітини, звані нейронами. Вони мають ряд спільних з клітинами інших типів рис, при цьому сильно відрізняючись від них за своєю конфігурацією та функціональним призначенням. Активність нейронів при передачі та обробці нервових імпульсів регулюється властивостями мембрани, які можуть змінюватись під впливом синаптичних медіаторів. Біологічні функції нейрона можуть змінюватись та адаптуватися до умов функціонування. Нейрони поєднуються в нейронні мережі, основні типи яких, а також схеми провідних шляхів мозку є генетично запрограмованими. У процесі розвитку можлива локальна видозміна нейронних мереж із формуванням нових сполук між нейронами. Відзначимо також, що нервова система містить, крім нейронів, клітини інших типів.

Історично першою роботою, що заклала теоретичний фундамент для створення штучних моделей нейронів та нейронних мереж, прийнято вважати опубліковану у 1943 р. статтю Уоррена С. Мак-Каллока і Вальтера

Піттса "Логічні обчислення ідей, що відносяться до нервової активності". Головний принцип теорії Маккалока і Піттса полягає в тому, що довільні явища, що відносяться до вищої нервової діяльності, можуть бути проаналізовані та зрозумілі як деяка активність у мережі, що складається з логічних елементів, що приймають лише два стани ("все або нічого"). При цьому для будь-якого логічного вираження, що задовольняє зазначеним авторами умовам, може бути знайдена мережа логічних елементів, що має поведінку, що описується цим виразом.

1. Ці ідеї через кілька років розвинув американський нейрофізіолог Френк Розенблатт. Він запропонував схему пристрою, що моделює процес людського сприйняття і назвав його *«перцептроном»* [2]. Перцептрон передавав сигнали від фотоелементів, що являють собою сенсорне поле, блоки електромеханічних осередків пам'яті. Ці осередки з'єднувалися між собою випадково відповідно до принципів коннективізму. Таким чином перцептрон, або персептрон – це математична або комп'ютерна модель сприйняття інформації мозком (кібернетична модель мозку), запропонована Френком Розенблаттом у 1957 році та вперше реалізована у вигляді електронної машини «Марк-1» у 1960 році. Перцептрон став однією з перших моделей нейромереж, а "Марк-1" – першим нейрокомп'ютером [2].

Отже, у якості моделі логічного елемента Мак-Каллока і Піттса, який отримав надалі назву *"формальний нейрон"*, була запропонована схема, наведена на рисунку 1.3. З сучасної точки зору, формальний нейрон є математичною моделлю простого процесора, що має кілька входів і один вихід. *Вектор входних сигналів* (що надходять через "дендрити") перетворюється нейроном у *вихідний сигнал* (що поширюється "аксоном") з використанням трьох функціональних блоків: локальної пам'яті, блоку підсумовування і блоку нелінійного перетворення.

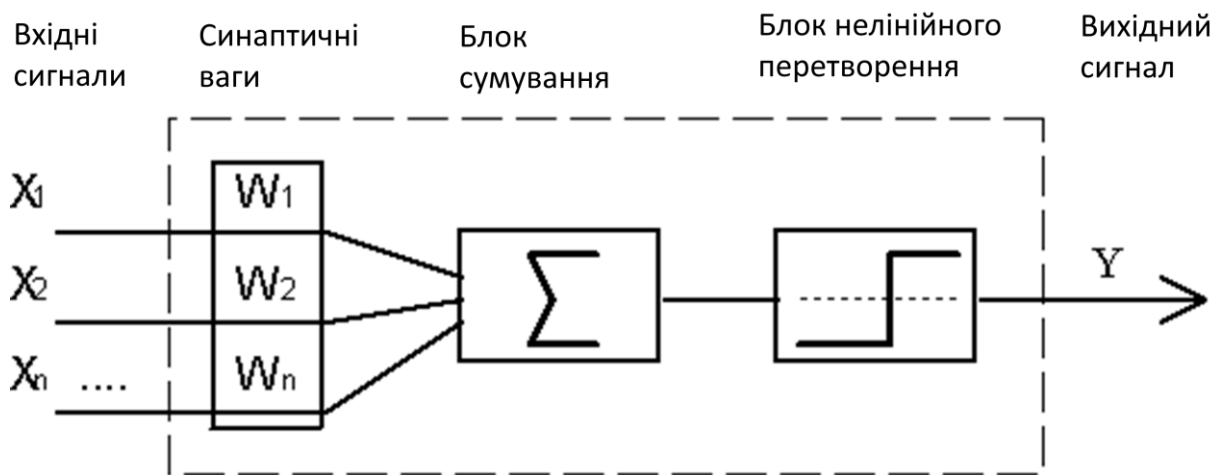


Рисунок 1.3 – Функціональна схема формального нейрона Маккалока та Піттса

Вектор локальної пам'яті містить інформацію про вагові множники, з якими вхідні сигнали інтерпретуватимуться нейроном. Ці змінні ваги є аналогом чутливості пластичних синаптичних контактів. Вибором цих ваг досягається та чи інша інтегральна функція нейрона[3-7].

У блоці підсумовування відбувається накопичення загального вхідного сигналу (результат позначимо символом **net**), що дорівнює зваженій сумі входів:

$$net = \sum_{j=1}^n W_j x_j .$$

У моделі Маккалока і Піттса відсутні тимчасові затримки вхідних сигналів, тому значення **net** визначає повне зовнішнє збудження, сприйняте нейроном. Відгук нейрона далі описується за принципом "все або нічого", тобто змінна проходить нелінійне порогове перетворення (функція активації), при якому вихід (стан активації нейрона) Y встановлюється рівним одиниці, якщо **net** > Поріг, і $Y = 0$ у протилежному випадку. Значення порога (часто рівне нулю) також зберігається в локальній пам'яті.

Формальні нейрони можуть бути об'єднані в мережі шляхом замикання виходів одних нейронів на входи інших, і на думку авторів моделі така кібернетична система з належно обраними вагами може представляти довільну логічну функцію. Для теоретичного опису нейронних мереж, що

виходять таким чином, пропонувалася математична мова обчислення логічних предикатів.

Слід зазначити, що сьогодні, через 50 років після роботи Маккалока і Піттса, вичерпної теорії синтезу логічних нейронних мереж із довільною функцією немає. Найбільш просунутими виявилися дослідження в галузі багатошарових систем та мереж із симетричними зв'язками. Більшість моделей спираються у своїй основі на різні модифікації формального нейрона.

Таким чином результат роботи нейрона – це функція активації взята від суми входів нейронів, помножена на ваги із додаванням зміщення (з-за того, що суматор є лінійною комбінацією входів X і ваг W , то вихід нейрона $Y = f(z) = f(X_1 * W_1 + \dots X_n W_n)$ – це насправді лінійна функція, а весь спектр лінійних функцій – це $y = kx + b$, де b – це зміщення, яке ми додаємо в суматорі).

Докладніше, що таке нейрон зсуву (використовується поняття функції активації, яке більш детально надано наприкінці розділу).

Нейрон зміщення (зсуву) або bias нейрон – це третій вид нейронів, що використовується в більшості нейромереж. Особливість цього типу нейронів полягає в тому, що його вхід і вихід завжди дорівнюють 1 і вони ніколи не мають вхідних синапсів. Нейрони зміщення можуть, або бути присутніми в нейронній мережі по одному на шарі, або повністю бути відсутніми. Проміжного стану бути не може (червоним на рисунку 1.4 позначені ваги і нейрони, які розміщувати не можна). З'єднання у нейронів зміщення такі ж, як у звичайних нейронів – з усіма нейронами наступного рівня, за винятком того, що синапсів між двома *bias* нейронами бути не може. Отже, їх можна розміщувати на вхідному шарі і всіх прихованих шарах, але ніяк не на вихідному шарі, тому що їм просто не буде з чим формувати зв'язок [8].

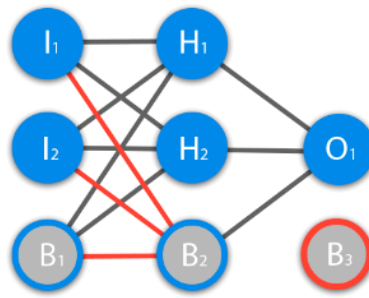


Рисунок 1.4 – Схематичне зображення нейронної мережі із нейроном зсуву та дозволеними та забороненими станами

Навіщо потрібен нейрон зміщення?

Нейрон зміщення потрібен для того, щоб мати можливість отримувати вихідний результат шляхом зсуву графіка функції активації вправо або вліво (рисунок 1.5). Розглянемо простий приклад, де є один вхідний нейрон і один вихідний нейрон.

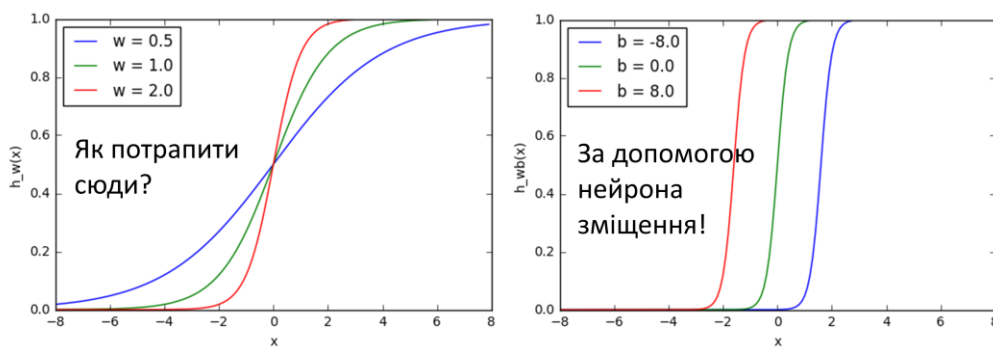


Рисунок 1.5 – Приклад використання нейрона зміщення

Зі шкільного курсу математики, ми знаємо, що якщо взяти функцію $y = a \cdot x + b$ і змінювати в неї значення “а”, то змінюватиметься нахил функції (колір ліній на лівому рисунку 1.5), а якщо змінювати “b”, то ми зміщуватимемо функцію вправо або вліво (кольори ліній на правому рисунку 1.5). Так ось “a” – це наші ваги, а “b” - це ваги нейрона зміщення. Це грубий приклад, але приблизно так все і працює. Тобто, коли під час навчання ми регулюємо ваги прихованих і вихідних нейронів, ми змінюємо нахил функції активації. Однак, регулювання ваги нейронів зміщення може дати нам можливість зрушити функцію активації по осі X і захопити нові

ділянки. Іншими словами, якщо точка, що відповідає за наше рішення, буде перебувати, як показано на рисунку 1.5 зліва, то наша нейронна мережа ніколи не зможе вирішити задачу без використання нейронів зміщення. Саме тому рідко можна зустріти нейронні мережі без нейронів зміщення.

Також нейрони зміщення допомагають у тому випадку, коли всі вхідні нейрони отримують на вхід 0 і незалежно від того які у них ваги, вони передадуть на наступний шар 0, але не у разі присутності нейрона зміщення. Наявність або відсутність нейронів зміщення – це *гіперпараметр*. Одним словом, ми самі повинні вирішити, чи потрібно нам використовувати нейрони зміщення чи ні, протестувавши нейронну мережу з нейронами зміщення і без них і порівнявши результати [1-8].

Важливо знати, що іноді на схемах не позначають нейрони зміщення, а просто враховують їх ваги при обчисленні вхідного значення, наприклад: $input = H1*w1 + H2*w2 + b3$, $b3 = bias*w3$. Так як його вихід завжди дорівнює 1, то можна просто уявити, що у нас є додатковий синапс з вагою і додати до суми цю вагу без згадки самого нейрона зміщення.

Таким чином, підсумовуючи, математична модель нейрона може бути записана як (f – функція активації):

$$Y = f(z) = f(X_1*W_1 + ...X_nW_n + b) = f(\sum_{i=1}^N w_i x_i + b) = f(\langle \underset{w}{\rightarrow}, \underset{x}{\rightarrow} \rangle + b).$$

Функції активації. Види функції активації.

Отже, що робить штучний нейрон? Простими словами, він розраховує зважену суму на своїх входах, додає зміщення (bias) і вирішує, чи слід це значення виключати або використовувати далі (так, функція активації так і працює, але давайте підемо по порядку).

Функція активації визначає вихідне значення нейрона в залежності від результату зваженої суми входів та порогового значення.

Розглянемо нейрон:

$$Y = \sum(weight * input) + bias$$

Тепер значення Y може бути будь-яким у діапазоні від мінус нескінченності до плюс нескінченності. Насправді у такій формі нейрон не знає межу, після якої слідує активація. Відповімо ж на питання, як ми вирішуємо, чи має бути нейрон активований (ми розглядаємо патерн активації, оскільки можемо провести аналогію з біологією. Саме таким чином працює мозок, а мозок – гарне свідчення роботи складної та розумної системи).

Для цього вирішили додавати **активаційну функцію**. Вона перевіряє вироблене нейроном значення Y щодо того, чи повинні зовнішні зв'язки розглядати цей нейрон як **активований**, чи його можна ігнорувати.

Ступінчаста функція активації.

Перше, що спадає на думку, це питання про те, що вважати межею активації для активаційної функції. Якщо значення Y більше деякого порогового значення, вважаємо нейрон активованим. Інакше говоримо, що нейрон неактивний. Така схема має спрацювати, але спочатку давайте її формалізуємо.

- Функція A = активована, якщо $Y >$ порогове значення, інакше ні.
- Інший спосіб: $A = 1$, якщо $Y >$ порогове значення, інакше $A = 0$.

Функція, яку ми щойно створили, називається *ступінчастою*. Така функція представлена на рисунку 1.6.

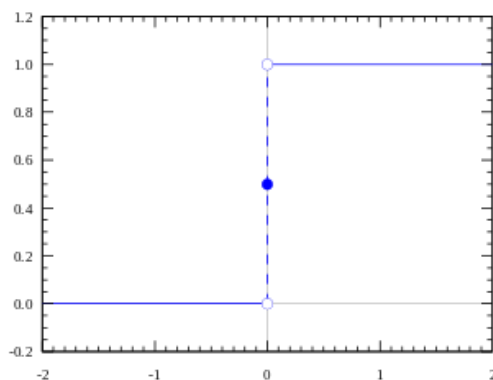


Рисунок 1.6 – Приклад ступінчастої функції активації

Функція приймає значення 1 (активована), коли $Y > 0$ (порогове значення), і значення 0 (не активована) інакше.

Ми створили активаційну функцію нейрону. Це простий спосіб, проте у цієї функції є недоліки. Розглянемо таку ситуацію.

Уявімо, що ми створюємо бінарний класифікатор – тобто у нас є модель, яка має говорити «так» чи «ні» («активований» чи «ні»). Ступінчаста функція зробить це за вас – вона точно виводить 1 або 0.

Тепер уявимо випадок, коли потрібна більша кількість нейронів для класифікації багатьох класів (багатокласовий класифікатор): клас1, клас2, клас3 і так далі. Що буде, якщо активованими виявляться більше ніж 1 нейрон? Усі нейрони з функції активації виведуть 1. У цьому разі виникає питання у тому, який клас має у результаті вийти для заданого об'єкта?

Ми хочемо, щоб активувався тільки один нейрон, а функції активації інших нейронів дорівнювали нулю (тільки в цьому випадку можна бути впевненим, що мережа правильно визначає клас). Таку мережу важче навчати та домагатися збіжності. Якщо активаційна функція не бінарна, то будуть можливі значення, на кшталт, активовано на 50%, активовано на 20% і так далі. Якщо активовано кілька нейронів, можна знайти нейрон з найбільшим значенням активаційної функції (краще, звичайно, щоб це була softmax функція (про неї поговоримо пізніше), а не max. Але поки не будемо займатися цими питаннями).

Але в такому випадку, як і раніше, якщо більше одного нейрона говорять "активований на 100%", проблема залишається. Так як існують проміжні значення на виході нейрона, процес навчання проходить більш гладко і швидко, а ймовірність появи кількох повністю активованих нейронів під час тренування знижується порівняно зі ступінчастою функцією активації (хоча це залежить від того, що ви навчаєте і на яких даних).

Отже, ми визначилися, що хочемо отримувати проміжні значення активаційної функції (аналогова функція), а не просто говорити активовано чи ні (бінарна функція).

Перше, що спадає на думку — це лінійна функція.

Лінійна функція активації.

$$A = cx$$

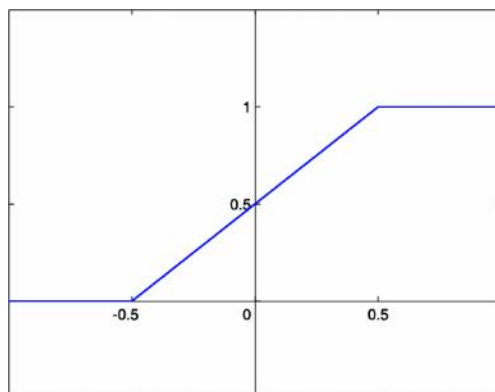


Рисунок 1.7 – Приклад лінійної функції активації

Лінійна функція є прямою лінією і пропорційна входу (тобто зваженій сумі на цьому нейроні, рисунок 1.7).

Такий вибір активаційної функції дозволяє отримувати спектр значень, а не лише бінарну відповідь. Можна з'єднати кілька нейронів разом і якщо більше одного нейрона активовано, рішення приймається на основі застосування операції $\max$ (або softmax). Але і тут не без проблем.

Якщо ви знайомі з методом градієнтного спуску для навчання, то можете помітити, що для цієї функції похідна дорівнює константі.

Похідна від $A=cx$ по x дорівнює c . Це означає, що градієнт не пов'язаний з x . Градієнт є постійним вектором, а градієнтний спуск здійснюється за постійним градієнтом. Якщо робиться помилкове передбачення, то зміни, зроблені зворотним поширенням помилки, теж постійні і залежить від зміни на вході $\text{delta}(x)$.

Це не добре (не завжди, але в більшості випадків). Але є й інша проблема. Розглянемо пов'язані шари. Кожен шар активується лінійною функцією. Значення з цієї функції йде в наступний шар як вхід, другий шар розраховує зважену суму на своїх входах і, у свою чергу, включає нейрони в залежності від іншої лінійної активаційної функції.

При цьому, не має значення скільки шарів ми маємо. Якщо всі вони за своєю природою лінійні, то фінальна функція активації в останньому шарі буде просто лінійною функцією від входів на першому шарі!

Це означає, що два шари (або N шарів) можуть бути замінені одним шаром. Тобто, використовуючи лінійну функцію активації, ми втратили можливість робити набори із шарів. Не важливо, як ми поєднуємо нейрони та шари, вся нейронна мережа все одно буде подібна до одного шару з лінійною функцією активації (тобто, можемо сказати, що комбінація лінійних функцій лінійним чином = інша лінійна функція).

Сигмоїдна функція активації.

$$A = \frac{1}{1 + e^{-x}}$$

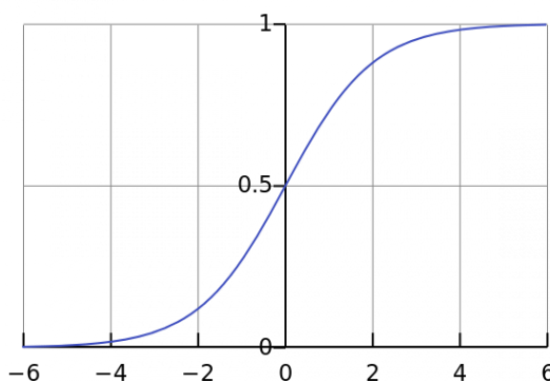


Рисунок 1.8 – Приклад сигмоїдної функції активації

Сигмоїда виглядає гладкою і подібна до ступінчастої функції. Розглянемо її переваги.

По-перше, сигмоїда – нелінійна за своєю природою, а комбінація таких функцій робить також нелінійну функцію. Отже, із нею ми можемо поєднувати шари.

Ще одна перевага такої функції – вона не бінарна, що робить активацію аналоговою, на відміну від ступінчастої функції. Для сигмоїди також характерний гладкий градієнт.

Як можна помітити з рисунка 1.8, у діапазоні значень X від -2 до 2 значення Y змінюється дуже швидко. Це означає, що будь-яка мала зміна значення X в цій області призводить до суттєвої зміни значення Y . Така поведінка функції вказує на те, що Y має тенденцію притискатись до одного з країв кривої.

Сигмоїда дійсно виглядає гарною функцією для класифікаційних завдань. Вона прагне привести значення до однієї зі сторін кривої (наприклад, до верхнього краю при $x = 2$ і нижнього краю при $x = -2$). Така поведінка дозволяє знаходити чіткі границі під час передбачення.

Інша перевага сигмоїди над лінійною функцією полягає в наступному. У першому випадку маємо фіксований діапазон значень функції $[0,1]$, тоді як лінійна функція змінюється в межах $(-\infty, \infty)$. Така властивість сигмоїди дуже корисна, тому що не призводить до помилок у разі великих значень активації.

Сигмоїда є однією з найчастіших активаційних функцій у нейромережах. Але й у неї є недоліки, на які варто звернути увагу.

Можна помітити, що при наближенні до кінців сигмоїди значення Y мають тенденцію слабо реагувати на зміни в значенні X . Це означає, що градієнт у таких областях набуває невеликих значень. А це, у свою чергу, призводить до проблеми *«зникнення градієнту» (vanishing gradient problem)*.

Розглянемо, що відбувається при наближенні активаційної функції до майже горизонтальної частини кривої на обох сторонах сигмоїди. У такому разі значення градієнта мале або зникає (не може зробити суттєвої зміни

через надзвичайно малі значення – проблема загасаючого градієнта). Нейромережа відмовляється навчатися далі або робить це вкрай повільно (залежно від способу використання або доти, доки градієнт/обчислення не почне страждати від обмежень на значення з плаваючою точкою).

Існують варіанти виправлення цих проблем, а сигмоїда все ще дуже популярна для класифікаційних завдань.

Гіперболічний тангенс.

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1 = 2\text{sigmoid}(2x) - 1$$

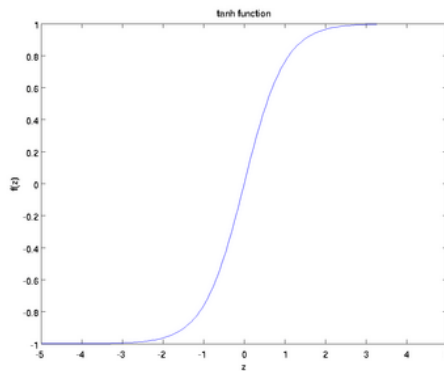


Рисунок 1.9 – Приклад функції активації гіперболічний тангенс

Ще одна функція активації, що часто використовується – гіперболічний тангенс. Гіперболічний тангенс дуже схожий на сигмоїду. Це скоригована сигмоїдна функція.

Тому така функція має ті самі характеристики, як і в сигмоїди, розглянутої раніше. Її природа нелінійна, добре підходить для комбінації шарів, а діапазон значень функції – $(-1, 1)$. Тому немає сенсу турбуватися, що активаційна функція перевантажиться від величезних значень. Однак варто відзначити, що градієнт тангенціальної функції більший, ніж у сигмоїди (похідна крутіша). Рішення про те, чи вибрати сигмоїду чи тангенс, залежить від вимог до амплітуди градієнта. Як і сигмоїди, гіперболічному тангенсу властива проблема зникнення градієнта.

Тангенс також є дуже популярною активаційною функцією.

Rectified linear unit (ReLU)

Наступна у нашому списку – активаційна функція ReLU (рисунок 1.10).

$$A(x) = \max(0, x)$$

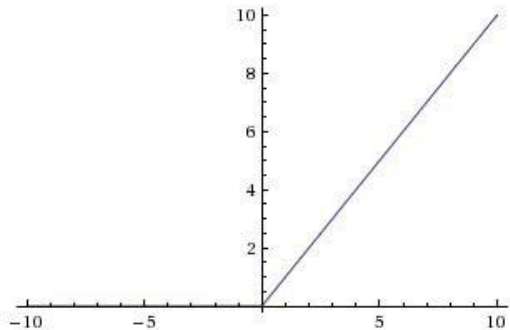


Рисунок 1.10 – Приклад функції активації ReLU

Насправді їх існує ціла родина таких функцій (RELU, Exponential RELU, Parametric RELU, Leaky RELU, Randomized RELU), але ми не будемо на них зупинятися, докладно розберемо тільки головну з них.

Користуючись визначенням, стає зрозуміло, що ReLU повертає значення x якщо x позитивно, і 0 в іншому випадку.

На перший погляд здається, що ReLU має ті самі проблеми, що й лінійна функція, оскільки ReLU лінійна в першому квадранті. Але насправді, ReLU нелінійна за своєю природою, а комбінація ReLU також нелінійна! (Насправді, така функція є хорошим апроксиматором, оскільки будь-яка функція може бути апроксимована комбінацією ReLU). Це означає, що ми можемо поєднувати шари. Область допустимих значень ReLU – $[0, \infty)$, тобто активація може "*вибухнути*".

Наступний пункт – розрідженість активації. Уявимо велику нейронну мережу з безліччю нейронів. Використання сигмоїди або гіперболічного тангенсу спричинить активацію всіх нейронів аналоговим способом. Це означає, що майже всі активації мають бути оброблені для опису виходу мережі. Інакше кажучи, активація щільна, але це затратно. В ідеалі ми

хочемо, щоб деякі нейрони не були активовані, це зробило б активації розрідженими та ефективними.

ReLU дозволяє це зробити. Уявимо мережу з випадково ініціалізованими вагами (або нормалізованими), в якій приблизно 50% активацій дорівнюють 0 через характеристики ReLU (повертає 0 для негативних значень x). У такій мережі включається менша кількість нейронів (розріджена активація), а мережа стає легше. Відмінно, здається, що ReLU підходить за всіма параметрами. Але ніщо не ідеальне, зокрема і ReLU.

Через те, що частина ReLU є горизонтальною лінією (для негативних значень x), градієнт на цій частині дорівнює 0. Через рівність нуля градієнта, ваги не коригуватимуться під час градієнтного спуску. Це означає, що нейрони, що знаходяться в такому стані, не будуть реагувати на зміни в помилці/вхідних даних (просто тому, що градієнт дорівнює нулю, нічого не змінюватиметься). Таке явище називається проблемою вмираючого ReLU (**Dying ReLU problem**). Через цю проблему деякі нейрони просто вимикаються і не відповідатимуть, роблячи значну частину нейромережі пасивною. Однак є варіації ReLU, які допомагають цю проблему уникнути. Наприклад, є сенс замінити горизонтальну частину функції на лінійну. Якщо вираз для лінійної функції визначається виразом $y = 0.01x$ для області $x < 0$, лінія злегка відхиляється від горизонтального положення. Існують інші способи уникнути нульового градієнта. Основна ідея – зробити градієнт нерівним нулю та поступово відновлювати його під час тренування.

ReLU менш вимоглива функція до обчислювальних ресурсів, ніж гіперболічний тангенс або сигмоїда, оскільки робить простіші математичні операції. Тому є сенс використовувати ReLU при створенні глибоких нейронних мереж.

Як вибрати функцію активації?

Настав час вирішити, яку функцію активації використовувати. Чи слід використовувати ReLU для кожного випадку? Чи сигмоїду? Або tanh? На ці питання не можна дати однозначної відповіді. Коли ви знаєте деякі характеристики функції, яку намагаєтеся апроксимувати, вибирайте активаційну функцію, яка апроксимує потрібну функцію краще і веде до більш швидкого навчання (наприклад, сигмоїда добре показує себе в задачах класифікації). Використовуйте функцію, з якою процес навчання та збіжність рішення будуть швидше. Більше того, ви можете використати власну функцію активації. Якщо ви не знаєте природу досліджуваної функції, то почніть з ReLU і потім працюйте у зворотному напрямку. У більшості випадків ReLU працює як хороший апроксиматор.

Нейронне представлення логічних вентилів AND, OR, NOT, XOR та XNOR (алгоритм перцептрона, часте питання на співбесідах)

Примітка. Мета цього розділу **не** полягає в тому, щоб математично пояснити, як нейронна мережа оновлює ваги, а в простому поясненні логіки зміни значень.

По-перше, нам потрібно згадати, що алгоритм роботи Перцептрона стверджує, що (використовується порогова функція активації):

$$\begin{aligned} \text{Прогноз } (y') &= 1, \text{ якщо } Wx + b > 0 \\ &\text{та } 0, \text{ якщо } Wx + b \leq 0 \end{aligned}$$

Кроки методу, який ми розглянемо (алгоритм Перцептрона), дуже схожі на те, як навчаються нейронні мережі, а саме нам необхідно:

- Ініціалізувати значення ваг та зміщення
- Пряме поширення (прямий розрахунок)
- Перевірити помилку

- Зворотне поширення (зворотний розрахунок) та коригування ваг та зміщення
- Повторити для всіх навчальних прикладів

Тепер, коли ми знаємо кроки, які необхідно зробити, почнемо працювати.

AND

Логічна таблиця оператора AND наведена на рисунку 1.11.

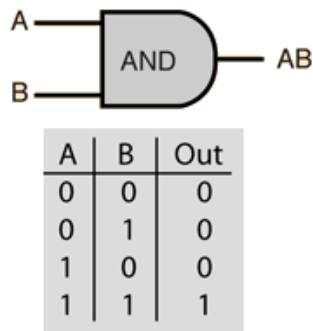


Рисунок 1.11 – Логічна таблиця оператора AND

Питання полягає в тому, які ваги та зміщення необхідно задати для перцептрону який буде виконувати функції AND?

По-перше, нам потрібно зрозуміти, що вихід логічного елемента AND дорівнює 1, тільки якщо обидва входи (в даному випадку x_1 і x_2) дорівнюють 1. Отже, слідуючи крокам, перерахованим вище отримуємо:

Ряд 1

- З $w_1 * x_1 + w_2 * x_2 + b$, ініціалізуючи w_1 , w_2 як 1 і b як -1 ми отримуємо $x_1*(1) + x_2*(1) -1$
- Проходячи перший рядок логічної таблиці ($x_1 = 0$, $x_2 = 0$), отримуємо: $0 + 0 -1 = -1$
- Відповідно до правила перцептрона, якщо $Wx + b \leq 0$, то $y' = 0$. Таким чином, цей рядок вірний, і у зворотному поширенні помилки немає необхідності.

Ряд 2 , 3

- Проходячи другий рядок ($x_1 = 0$ та $x_2 = 1$), отримуємо: $0 + 1 -1 = 0$

- Відповідно до правила перцептрона, якщо $Wx + b \leq 0$, то $y = 0$. Цей рядок вірний, тому що на виході логічного елемента AND буде 0.
- Відповідно до правила перцептрону це працює (як для рядка 1, так і для рядка 2 та 3).

Ряд 4

- Переходячи до останнього рядка ($x_1 = 1$ та $x_2 = 1$), отримуємо: $1+1-1 = 1$
- Знову ж таки, виходячи з правила перцептрону, все ще в силі.

Отже, ми можемо зробити висновок, що модель для досягнення логічного елемента AND з використанням алгоритму перцептрону може бути задана наступним чином: $x_1 + x_2 - 1$:

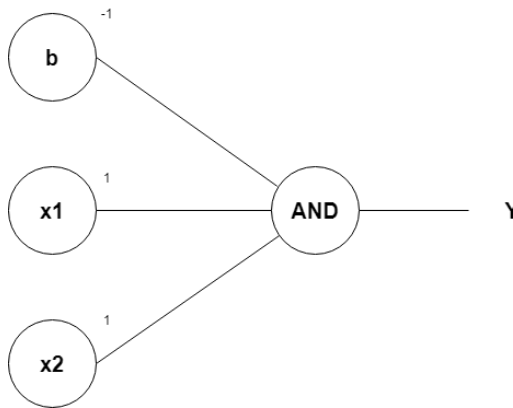
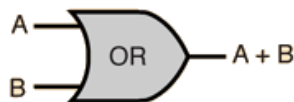


Рисунок 1.12 – Модель логічного елемента AND

OR

Логічна таблиця елемента OR наведена на рисунку 1.13.



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

Рисунок 1.13 – Логічна таблиця елемента OR

На схемі логічний елемент OR дорівнює 0, тільки якщо обидва входи дорівнюють 0. Робимо аналогічні дії.

Ряд 1

- З $w_1x_1 + w_2x_2 + b$, ініціалізуючи w_1 , w_2 як 1 і b як -1 ми отримуємо: $x_1(1) + x_2(1) - 1$
- Проходячи перший рядок таблиці елементу OR ($x_1 = 0$, $x_2 = 0$), отримуємо: $0 + 0 - 1 = -1$
- Відповідно до правила перцептрона, якщо $Wx + b \leq 0$, то $y' = 0$. Отже, цей рядок вірний.

Ряд 2

- Проходячи наступний рядок ($x_1 = 0$ та $x_2 = 1$), отримуємо: $0 + 1 - 1 = 0$
- Відповідно до правила перцептрона, якщо $Wx + b \leq 0$, то $y' = 0$. Отже, цей рядок неправильний.
- Отже, нам потрібні значення, які змусять вхідні дані $x_1 = 0$ і $x_2 = 1$ дати у значення 1. Якщо ми змінимо w_2 на 2, ми отримаємо: $0 + 2 - 1 = 1$
- Відповідно до правила перцептрону, це правильно як для рядка 1, так і для рядка 2.

Ряд 3

- Переходячи ($x_1 = 1$ та $x_2 = 0$), отримуємо: $1 + 0 - 1 = 0$
- Відповідно до правила перцептрона, якщо $Wx + b \leq 0$, то $y' = 0$. Отже, цей рядок неправильний.
- Оскільки він аналогічний рядку 2 ми можемо просто змінити w_1 на 2, ми маємо: $2 + 0 - 1 = 1$
- Відповідно до правила перцептрону, це правильно як для рядків 1, 2 та 3.

Ряд 4

- Проходячи ($x_1 = 1$ та $x_2 = 1$), отримуємо: $2 + 2 - 1 = 3$

- Знову ж таки, виходячи з правила перцептрону, це все ще в силі.

Отже, ми можемо зробити висновок, що модель логічного елемента OR з використанням алгоритму перцептрон може бути задана наступним чином: $2x_1 + 2x_2 - 1$

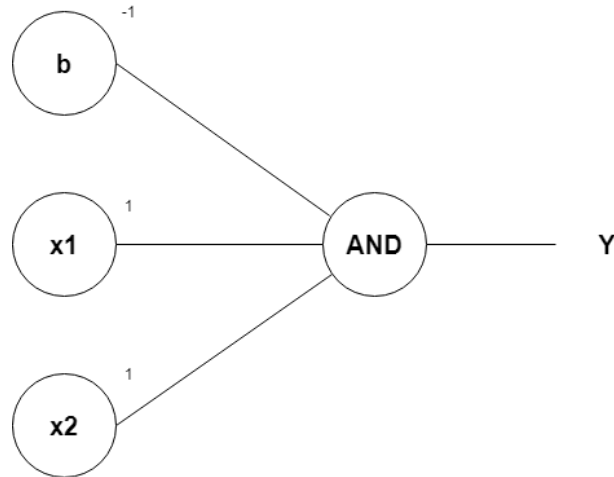


Рисунок 1.14 – Модель логічного елемента OR

NOT

Логічна таблиця елемента NOT наведена на рисунку 1.15.

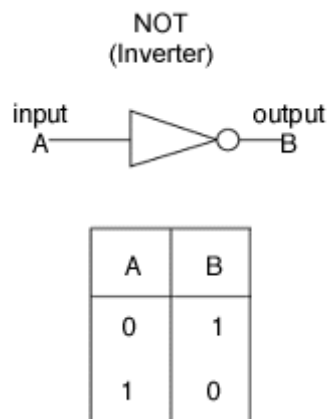


Рисунок 1.15 – Логічна таблиця елемента OR

Як очевидно зі схеми, вихід логічного елемента NOT є інверсією одиночного входу. Отже, відповідно до алгоритму наведеного вище маємо:

Ряд 1

- З $w_1x_1 + b$, ініціалізуючи w_1 як 1 (з єдиного входу) і b як -1 ми отримуємо: $x_1 (1) - 1$

- Пройшовши перший рядок логічної таблиці NOT ($x_1 = 0$) отримуємо: $0 - 1 = -1$
- Відповідно до правила перцептрона, якщо $Wx + b \leq 0$, то $y' = 0$. Отже, цей рядок неправильний, тому що для логічного елемента NOT на виході буде 1.
- Отже, нам потрібні значення, які зроблять вхід $x_1 = 0$, щоб дати у значення 1. Якщо ми змінимо b на 1, ми отримаємо: $0 + 1 = 1$
- За правилом перцептрону це тепер працює.

Ряд 2

- Проходячи ($x_1 = 1$), отримуємо: $1 + 1 = 2$
- Відповідно до правила перцептрону, якщо $Wx + b > 0$, то $y' = 1$. Отже, цей рядок невірний, тому що для логічного елемента NOT на виході буде 0.
- Отже, нам потрібні значення, які зроблять вхід $x_1 = 1$, щоб дати у значення 0. Якщо ми змінимо w_1 на -1 , ми отримаємо: $-1 + 1 = 0$
- Відповідно до правила перцептрону, якщо $Wx + b \leq 0$, то $y' = 0$. Отже, це працює (як для рядка 1, так і для рядка 2).
- Отже, ми можемо зробити висновок, що модель задля досягнення логічного елемента NOT з використанням алгоритму перцептрону може мати наступний вигляд: $-x_1 + 1$

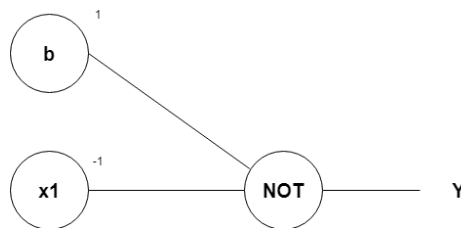


Рисунок 1.16 – Модель логічного елемента NOT

NOR

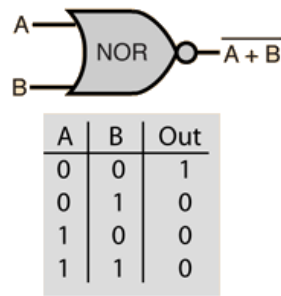


Рисунок 1.17 – Логічна таблиця елементу NOR

Логічний вентиль NOR дорівнює 1, тільки якщо обидва входи дорівнюють 0.

Ряд 1

- З $w_1x_1 + w_2x_2 + b$, ініціалізуючи w_1 і w_2 як 1, а b як -1 ми отримуємо: $x_1(1) + x_2(1) - 1$
- Пройшовши перший рядок логічної таблиці NOR ($x_1 = 0, x_2 = 0$), отримуємо: $0 + 0 - 1 = -1$
- Відповідно до правила перцептрону, якщо $Wx + b \leq 0$, то $y' = 0$. Цей рядок неправильний, оскільки вихід дорівнює 1 для логічного елемента NOR.
- Отже, нам потрібні значення, які зроблять вхід $x_1 = 0$ та $x_2 = 0$, щоб дати у значення 1. Якщо ми змінимо b на 1, ми отримаємо: $0 + 0 + 1 = 1$
- За правилом перцептрон тепер все працює.

Ряд 2

- Переходячи до наступного рядку ($x_1 = 0, x_2 = 1$), отримуємо: $0 + 1 + 1 = 2$
- Відповідно до правила перцептрону, якщо $Wx + b > 0$, то $y' = 1$. Цей рядок неправильний, тому що для логічного елемента NOR на виході буде 0.

- Отже, хочемо, щоб значення, у яких вхідні дані $x_1 = 0$ і $x_2 = 1$ давали y' значення 0. Якщо ми, наприклад, змінимо w_2 на -1 , отримаємо: $0 - 1 + 1 = 0$
- Згідно з правилом перцептрону, це дійсно як для рядка 1, так і для рядка 2.

Ряд 3

- Переходячи до наступного рядка ($x_1 = 1, x_2 = 0$), отримуємо: $1 + 0 + 1 = 2$
- Відповідно до правила перцептрону, якщо $Wx + b > 0$, то $y' = 1$. Цей рядок неправильний, тому що для логічного елемента NOR на виході буде 0.
- Таким чином, нам потрібні значення, які робитимуть вхід $x_1 = 0$ та $x_2 = 1$, щоб дати y' значення 0. Якщо ми змінимо w_1 на -1 , ми отримаємо: $-1 + 0 + 1 = 0$
- Згідно з правилом перцептрону, це дійсно для рядків 1, 2 та 3.

Ряд 4

- Переходячи до останнього рядка ($x_1 = 1, x_2 = 1$), отримуємо: $-1 - 1 + 1 = -1$
- Згідно з правилом перцептрону все працює.

Отже, ми можемо зробити висновок, що модель для досягнення логічного елемента NOR з використанням алгоритму перцептрону може мати наступний вигляд: $-x_1 - x_2 + 1$

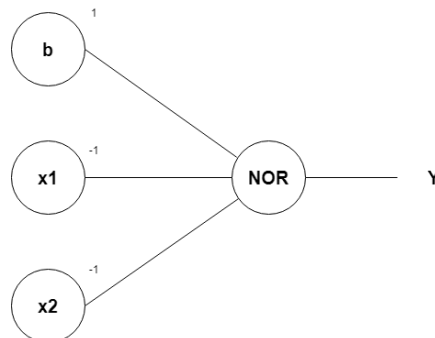


Рисунок 1.18 – Модель логічного елемента NOR

NAND

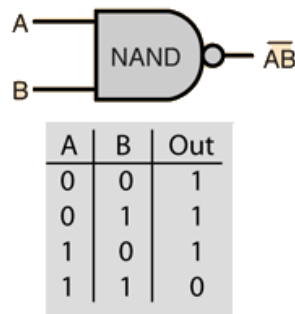


Рисунок 1.19 – Логічна таблиця елементу NAND

Логічний елемент NAND дорівнює 0, тільки якщо обидва входи дорівнюють 1.

Ряд 1

- З $w_1x_1 + w_2x_2 + b$, ініціалізуючи w_1 і w_2 як 1, а b як -1 ми отримуємо: $x_1 (1) + x_2 (1) - 1$
- Передавши перший рядок таблиці NAND ($x_1 = 0$, $x_2 = 0$), отримуємо: $0 + 0 - 1 = -1$
- Відповідно до правила перцептрону, якщо $Wx + b \leq 0$, то $y' = 0$. Цей рядок неправильний, оскільки вихід дорівнює 1 для логічного елемента NAND.
- Отже, нам потрібні значення, які зроблять вхід $x_1 = 0$ і $x_2 = 0$, щоб дати у значення 1. Якщо ми змінимо b на 1, ми отримаємо: $0 + 0 + 1 = 1$
- За правилом перцептрону все працює.

Ряд 2 , 3

- Переходячи до наступних рядків ($x_1 = 0$, $x_2 = 1$), отримуємо: $0 + 1 + 1 = 2$
- Відповідно до правила перцептрону, якщо $Wx + b > 0$, то $y' = 1$. Цей рядок також правильний (як для рядка 2, так і для рядка 3).

Ряд 4

- Переходячи до останнього рядка ($x_1 = 1, x_2 = 1$), отримуємо: $1 + 1 + 1 = 3$
- Це неправильний результат, оскільки вихід дорівнює 0.
- Змінивши значення w_1 та w_2 на -1, а значення b на 2, ми отримаємо: $-1-1 + 2 = 0$
- Тепер все працює для всіх рядків.

Таким чином ми можемо зробити висновок, що модель для досягнення логічного елемента NAND з використанням алгоритму перцептрону може виглядати таким чином: $-x_1-x_2 + 2$

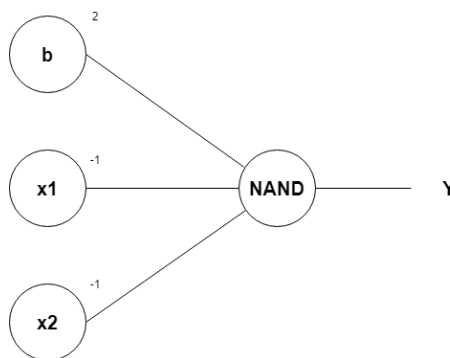


Рисунок 1.20 – Модель логічного елемента NAND

XNOR

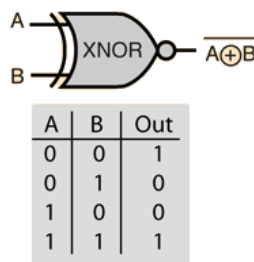


Рисунок 1.21 – Логічна таблиця елемента XNOR

Тепер коли ми закінчили з необхідними основними логічними вентилями, ми можемо об'єднати їх, щоб отримати вентиль XNOR.

Логічне уявлення елемента XNOR наступне: $x_1x_2 + x_1'x_2'$

Де ' - означає зворотне значення (invers) .

З наведеного виразу можна сказати, що вентиль XNOR складається з логічного елемента AND (x_1x_2), елемента NOR ($x_1'x_2'$) та логічного елемента OR.

Це означає, що нам доведеться об'єднати 3 перцептрони:

- AND ($x_1 + x_2 - 1$)
- NOR ($-x_1 - x_2 + 1$)
- OR ($2x_1 + 2x_2 - 1$)

Користуючись правилом перцептрону, ми можемо зробити висновок, що модель для досягнення логічного елемента XNOR з використанням алгоритму перцептрону може виглядати як показано на рисунку 1.21.

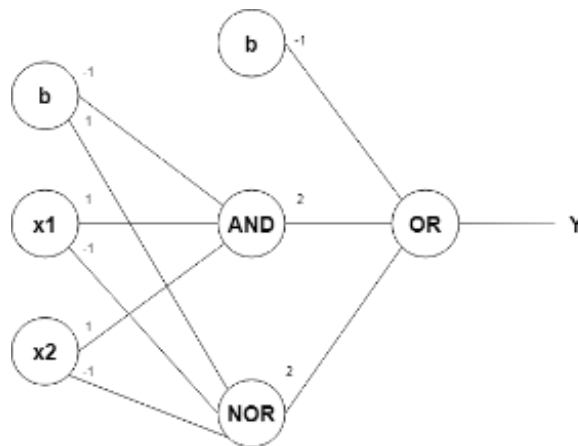


Рисунок 1.21 – Модель логічного елемента XNOR

XOR

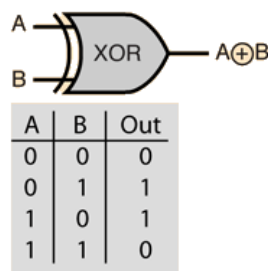


Рисунок 1.22 – Логічна таблиця елемента XOR

Логічне уявлення елемента XOR наступне: $x_1x_2' + x_1'x_2$

Спочатку спростимо логічний вираз:

$$x_1'x_2 + x_1x_2' + x_1'x_1 + x_2'x_2 = x_1(x_1' + x_2') + x_2(x_1' + x_2') = (x_1 + x_2)(x_1' + x_2') = (x_1 + x_2)(x_1x_2)'$$

Зі спрощеного виразу ми можемо сказати, що вентиль XOR складається з логічного елемента OR ($x_1 + x_2$), логічного елемента NAND ($-x_1 - x_2 + 1$) та логічного елемента AND ($x_1 + x_2 - 1,5$).

Це означає, що нам доведеться об'єднати 2 перцептрони:

- OR ($2x_1 + 2x_2 - 1$)
- NAND ($-x_1 - x_2 + 2$)
- AND ($x_1 + x_2 - 1$)

Користуючись правилом перцептрону, ми можемо зробити висновок, що модель для досягнення логічного елемента XOR з використанням алгоритму перцептрону може виглядати як показано на рисунку 1.23.

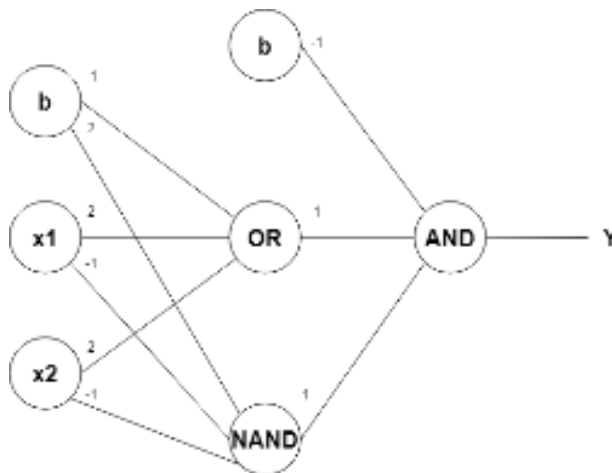


Рисунок 1.23 – Модель логічного елементу XOR

Зауважимо, що наведені вище моделі - це просто один із способів досягнення даного результату. Є багато інших способів та значень, які можна було б використовувати для досягнення потрібних логічних операторів з використанням перцептронів. Наприклад:

- AND ($20x_1 + 20x_2 - 30$)
- OR ($20x_1 + 20x_2 - 10$)
- NOT ($-20x_1 + 10$)

ТЕМА №2: НАВЧАННЯ НЕЙРОННОЇ МЕРЕЖІ. ФУНКЦІЯ ВТРАТ ЗАДАЧІ РЕГРЕСІЇ. ГЛИБОКА НЕЙРОННА МЕРЕЖА ТА ЇХ РІЗНОВИДИ.

Етапи життєвого циклу штучної нейронної мережі (ШНМ).

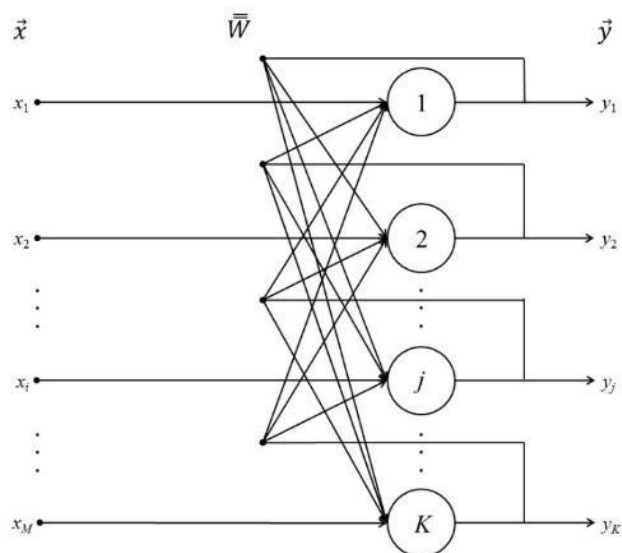


Рисунок 2.1 – Приклад архітектури штучної нейронної мережі

Використання штучної нейронної мережі для вирішення прикладних завдань – трудомісткий, а у деяких випадках і тривалий процес. У цьому зв'язку можна назвати і описати кілька загальних етапів її життєвого циклу [9].

1. Постановка задачі та вибір архітектури нейронної мережі

На даному етапі необхідно отримати чітке розуміння про задачу, що вирішується, причини використання для цього методу нейромережевого моделювання і переваги, які він повинен дати. При цьому, варто розуміти, що під конкретну постановку завдання підбирається архітектура нейронної мережі.

2. Визначення кількісного та якісного складів входів та виходів

Якісний склад вхідних та вихідних змінних нейромережевої моделі визначається, головним чином, постановкою завдання, що було зроблено на

попередньому кроці. Кількісний склад може залежати ще й від природи вхідних та вихідних змінних та процесів та властивостей, що ними описуються. Так, при необхідності масштабування деяких змінних кількості фактичних входів та виходів нейронної мережі може збільшитися порівняно з кількістю входів та виходів моделі.

3. Формування вихідної вибірки даних

Вихідна вибірка даних може бути сформована з урахуванням практичного чи обчислювального експерименту. Чим більший обсяг проведеного експерименту, тим більш точну нейромережну модель можна отримати та використовувати на практиці. Слід також враховувати вимоги до вибірки вихідних даних, сформульовані на попередньому кроці.

4. Попередня обробка та нормалізація вихідної вибірки

Даний етап визначається обраною архітектурою нейронної мережі, складом вхідних та вихідних змінних та, власне, фактичними даними, отриманими в результаті експерименту.

5. Поділ вихідної вибірки на навчальну та тестову складові

Наявна вихідна вибірка ділиться на навчальну та тестову вибірки з урахуванням архітектури нейронної мережі, що буде використовуватися. Зазвичай обсяг навчальної бази даних в кілька разів більший, ніж тестовий. Перша (навчальна) використовується строго для налаштування вагових коефіцієнтів. Друга (тестова) – для перевірки коректності налаштованої нейромережевої моделі. У разі обмеженого обсягу вихідної вибірки вона може бути задіяна для навчання, а оцінка коректності роботи мережі перевіряється на етапі практичної експлуатації.

6. Визначення структури нейронної мережі

Крім раніше встановленого складу вхідних та вихідних змінних на даному етапі задається кількість прихованих шарів та нейронів у кожному шарі. Для багатьох архітектур нейронних мереж вибір структури залежить, зокрема, від обсягу навчальних даних.

7. Налаштування параметрів нейронної мережі та алгоритму її навчання

На даному етапі задаються вид та параметри активаційних функцій прихованих та вихідних нейронів. Для ітераційного алгоритму вибираються умови закінчення навчання, коефіцієнт швидкості навчання, порядок пред'явлення прикладів навчальної вибірки.

8. Навчання нейронної мережі

Для точних алгоритмів зазвичай здійснюється одноразовий розрахунок вагових коефіцієнтів. Для ітераційних – багаторазове повторення епох навчання. Для навчання можуть бути використані лише приклади навчальної бази даних.

9. Контрастування нейронної мережі

Якщо алгоритм навчання нейронної мережі базувався на ітераційному налаштуванні вагових коефіцієнтів протягом усіх епох навчання, нерідкі випадки, коли деякі з ваг за абсолютним значенням настільки близькі до нуля, що не мають ніякого практичного впливу на результати розрахунків. Проте, такі розрахунки витрачають чимало обчислювальних ресурсів, що помітно уповільнює процес навчання. Для вирішення цієї проблеми такі вагові коефіцієнти повністю обнуляються, тобто відповідний зв'язок між входами та нейронами або між двома нейронами виключається із структури мережі.

10. Тестування нейронної мережі

Тестування нейронної мережі провадиться з використанням прикладів тестової вибірки. За результатами тестування оцінюється помилка роботи нейромережевої моделі.

Етапи 7-9 можуть виконуватися в циклі неодноразово, доки не буде отримана налаштована нейронна мережа, за допомогою якої можна вирішувати задачу з необхідним рівнем помилки.

11. Практичне використання

Навчена та протестована нейронна мережа використовується для практичного вирішення задачі у міру надходження нових вихідних даних (значень вхідних змінних). Результати з навченої мережі можуть бути отримані практично миттєво та інтерпретовані користувачем для подальшого прийняття рішень.

12. Донавчання нейронної мережі

Пари вхідних і вихідних векторів, що отримуються в ході практичної експлуатації нейронної мережі, можуть використовуватися для подальшого підстроювання вагових коефіцієнтів. Це особливо важливо, якщо спочатку обсяг вибірки даних був невеликим. Частина архітектур нейронних мереж – які використовують принцип самоорганізації – у ході донавчання можуть не тільки змінювати значення вагових коефіцієнтів, а й змінювати свою структуру.

Слід зауважити, що перерахована послідовність етапів характерна для роботи зі штучними нейронними мережами. Однак для ряду архітектур можуть мати місце властиві лише їм відмінності.

Далі розглянемо дещо спрощену модель побудови нейронної мережі, та більш детально проаналізуємо окремі етапи життєвого циклу ШНМ [8-10].

Скорочена модель побудови нейронної мережі

Наведені вище етапи можна скоротити до 5. При побудові моделі ШНМ передусім необхідно точно визначити задачі, які вирішуватимуться з її допомогою. В даний час технології нейронних мереж успішно застосовуються для *прогнозування, розпізнавання та узагальнення даних*.

Першим етапом побудови нейромережевої моделі є ретельний відбір вхідних даних, що впливають на очікуваний результат. З вихідної інформації необхідно виключити всі відомості, які не стосуються досліджуваної проблеми. У той же час слід мати достатню кількість прикладів для навчання ШНМ. Існує емпіричне правило, яке встановлює

рекомендоване співвідношення X між кількістю навчальних прикладів, що містять вхідні дані та правильні відповіді, і числом з'єднань у нейронній мережі: $X < 10$.

Для факторів, які включаються до навчальної вибірки, доцільно попередньо оцінити їх значущість, провівши кореляційний та регресійний аналіз та проаналізувати діапазони їх можливих змін.

На другому етапі здійснюється перетворення вихідних даних з урахуванням характеру та типу проблеми, що відображається моделлю, та вибираються способи подання інформації. Ефективність нейромережевої моделі підвищується, якщо діапазони зміни вхідних та вихідних величин приведені до деякого стандарту, наприклад $[0,1]$ або $[-1,1]$.

Третій етап полягає у конструюванні ШНМ, тобто у проектуванні її архітектури (кількість шарів і число нейронів у кожному шарі). Структура ШНМ формується до початку навчання, тому успішне вирішення цієї проблеми багато в чому визначається досвідом та мистецтвом аналітика, який проводить дослідження.

Четвертий етап пов'язаний з навчанням мережі, що може проводитися з урахуванням *конструктивного* чи *деструктивного* підходу. Відповідно до *першого підходу* навчання ШНМ – починається на мережі невеликого розміру, який поступово збільшується до досягнення необхідної точності за результатами тестування. *Деструктивний підхід* базується на принципі «проріджування дерева», відповідно до якого з мережі із завідомо надмірним обсягом нейронів поступово видаляють «зайві» нейрони та зв'язки, що примикають до них. Цей підхід дає можливість вивчити вплив віддалених зв'язків на точність мережі. Процес навчання нейронної мережі являє собою уточнення значень вагових коефіцієнтів для окремих вузлів на основі поступового збільшення обсягу вхідної та вихідної інформації. Початку навчання повинна передувати процедура вибору функції активації нейронів, що враховує характер розв'язуваної задачі. Зокрема, у тришарових перцептронах на нейронах прихованого шару застосовується в більшості

випадків логістична функція, а тип передавальної функції нейронів вихідного шару визначається на основі аналізу результатів обчислювальних експериментів на мережі. Індикатором навчання ШНМ може служити гістограма значень міжнейронних зв'язків.

На п'ятому етапі проводиться тестування отриманої моделі ШНМ на незалежній вибірці прикладів.

Навчання нейронних мереж

Найважливішою властивістю нейронних мереж є їх здатність до навчання, що робить такі моделі незамінними при вирішенні завдань, для яких алгоритмізація є неможливою, проблематичною або надто трудомісткою. Навчання нейронної мережі полягає у зміні внутрішніх параметрів моделі таким чином, щоб на виході ШНМ генерувався вектор значень, що збігається з результатами навчальної вибірки. Зміна параметрів нейромережевої моделі може виконуватися різними способами відповідно до різних алгоритмів навчання. Парадигма навчання визначається доступністю необхідної інформації. У цілому нині, основна маса завдань, розв'язуваних з допомогою методів машинного навчання, належить до різних видів навчання: *навчання з учителем (supervised learning)* чи *без нього (unsupervised learning)*. Однак цим учителем зовсім не обов'язково є сам програміст, який стоїть над комп'ютером та контролює кожну дію у програмі. «Вчитель» у термінах машинного навчання – це саме втручання людини у процес обробки інформації. В обох видах навчання машині надаються вихідні дані, які вона має проаналізувати і знайти закономірності. Відмінність лише у тому, що у навчанні з учителем є низка гіпотез, які потрібно спростувати чи підтвердити [10-12].

Загалом, виділяють наступні парадигми навчання:

- навчання з учителем (контрольоване);
- навчання без учителя (неконтрольоване);
- змішане навчання;

- навчання з підкріпленням (reinforcement learning)

Під час навчання з учителем всі приклади навчальної вибірки містять правильні відповіді (виходи), що відповідають вихідним даним (входам). У процесі контрольованого навчання синаптичні ваги налаштовуються так, щоб мережа породжувала відповіді, найбільш близькі до правильних.

Навчання без вчителя використовується, коли не для всіх прикладів навчальної вибірки відомі правильні відповіді. У цьому випадку робляться спроби визначення внутрішньої структури даних, що надходять до мережі, з метою розподілити зразки за категоріями (моделі Кохонена).

Різницю між цими двома парадигмами легко зрозуміти на простих прикладах.

Машинне навчання з учителем.

Припустимо, у нашому розпорядженні виявилися відомості про десять тисяч квартир: площа, поверх, район, наявність або відсутність паркування біля будинку, відстань від метро, ціна квартири і т.п. Нам необхідно створити модель, яка передбачає ринкову вартість квартири за її параметрами. Це ідеальний приклад машинного навчання з учителем: у нас є вихідні дані (кількість квартир та їх властивості, які називаються ознаками) та готова відповідь по кожній із квартир – її вартість. Програма має вирішити *завдання регресії*. Ще приклад із практики: підтвердити чи спростувати наявність раку у пацієнта, знаючи всі його медичні показники. З'ясувати, чи є вхідний лист спамом, проаналізувавши його текст. Це *завдання на класифікацію* [4-6].

Машинне навчання без вчителя.

У разі навчання без вчителя, коли готових «правильних відповідей» системі не надано, все ще цікавіше. Наприклад, у нас є інформація про вагу та зростання якоїсь кількості людей, і ці дані потрібно розподілити по трьох групах, для кожної з яких потрібно пошити сорочки відповідних розмірів.

Це завдання кластеризації. У цьому випадку доведеться розділити всі дані на 3 кластери (але, як правило, такого суворого і єдиного можливого поділу немає).

Якщо взяти іншу ситуацію, коли кожен з об'єктів у вибірці має сотню різних ознак, то основною складністю буде графічне відображення такої вибірки. Тому кількість ознак зазвичай зменшують до двох або трьох, і стає можливим візуалізувати їх на площині або у 3D. *Це завдання зменшення розмірності.*

Змішане навчання.

При змішаному навчанні частина ваг визначається за допомогою навчання з учителем, а інша частина за допомогою алгоритмів самонавчання.

Навчання за прикладами характеризується трьома основними властивостями: ємністю, складністю зразків та обчислювальною складністю. Місткість відповідає кількості зразків, які може запам'ятати мережа. Складність зразків визначає можливості нейронної мережі до навчання. Зокрема, при навчанні ШНМ можуть виникати стани «перетренування», в яких мережа добре функціонує на прикладах навчальної вибірки, але не справляється з новими прикладами, втрачаючи здатність навчатися.

Навчання із підкріпленням.

Існує ще такий цікавий метод, як навчання з підкріпленням (reinforcement learning). Такий спосіб навчання застосуємо тоді, коли ми можемо, ґрунтуючись на результатах отриманих від ШНМ, дати їй оцінку. Наприклад ми хочемо навчити НМ грати у якусь гру, де треба зібрати якомога більше очок. Тоді, щоразу коли НМ буде набирати багато очок ми будемо її заохочувати. Іншими словами ми надаємо НМ право знайти будь-який спосіб досягнення мети, доки він буде давати хороший результат. Таким чином, мережа почне розуміти, чого від неї хочуть досягти і

намагається знайти найкращий спосіб досягнення цієї мети без постійного надання даних “учителем”.

Загальні відомі правила навчання ШНМ.

Правило корекції за помилкою. Процес навчання ШНМ полягає в корекції вихідних значень вагових коефіцієнтів міжнейронних зв'язків, які зазвичай задаються випадковим чином. При введенні вхідних даних прикладу, що запам'ятовується (стимулу), з'являється реакція, яка передається від одного шару нейронів до іншого, досягаючи останнього шару, де обчислюється результат. Різниця між відомим значенням результату та реакцією мережі відповідає величині помилки, яка може використовуватися для коригування ваг міжнейронних зв'язків. Коригування полягає у невеликому (зазвичай менше 1%) збільшенні синаптичних ваг тих зв'язків, які посилюють правильні реакції, та зменшенні тих, що сприяють помилковим діям. Це найпростіше правило контрольованого навчання (дельта-правило) використовується в одношарових мережах з одним рівнем зв'язків, що налаштовуються, між безліччю входів і безліччю виходів. При цьому на кожному k -му кроці для j -го нейрона вага i -го зв'язку обчислюється за формулою $w_{jik} = w_{ji(k-1)} + \Delta w_{jik}$ де $\Delta w_{jik} = \eta \delta_{jk} x_{jik}$, $\delta_{jk} = T_{jk} - R_{jk}$, де T_{jk} – це відоме (правильне) значення виходу j -го нейрону; R_{jk} – розраховане значення виходу j -го нейрона; x_{jik} – величина сигналу на i -му вході, η – коефіцієнт швидкості навчання [5, 12-14].

Оптимальні значення ваг міжнейронних зв'язків можна визначити шляхом мінімізації середньоквадратичної помилки з використанням детермінованих або псевдовипадкових алгоритмів пошуку екстремуму в просторі вагових коефіцієнтів. При цьому виникає традиційна проблема оптимізації, що пов'язана з потраплянням у локальний мінімум.

Правило Хебба. Воно базується на наступному нейрофізіологічному спостереженні: якщо нейрони по обидва боки синапсу активізуються

одночасно і регулярно, то сила їхнього синаптичного зв'язку зростає. При цьому зміна ваги кожного міжнейронного зв'язку залежить тільки від активності нейронів, що утворюють синапс. Це значно спрощує реалізацію алгоритмів навчання.

Навчання методом змагання. На відміну від правила Хебба, де безліч вихідних нейронів можуть збуджуватися одночасно, у цьому випадку вихідні нейрони змагаються (конкурують) між собою за активізацію. У процесі змагального навчання здійснюється модифікація ваг зв'язків нейрона, що виграв, і нейронів, розташованих у його околиці («переможець забирає все»).

Метод зворотного поширення помилки. Він є узагальненням процедури навчання простого перцептрону з використанням дельта-правила на багат шарові мережі. У цьому методі необхідно мати навчальну вибірку, що містить «правильні відповіді», тобто вибірка повинна включати безліч пар зразків вхідних та вихідних даних, між якими потрібно встановити відповідність [3-6]. Перед початком навчання міжнейронним зв'язкам (вагам) надаються невеликі випадкові значення. Кожен крок навчальної процедури складається із *двох фаз*. Під час *першої фази* вхідні елементи мережі встановлюються у заданий стан. Вхідні сигнали поширюються мережею, породжуючи деякий вихідний вектор. Для роботи алгоритму потрібно, щоб характеристика вхід-вихід нейроподібних елементів була незменшувальною і мала обмежену похідну. Зазвичай при цьому використовують сигмоїдні функції. Отриманий вихідний вектор порівнюється з потрібним (правильним). Якщо вони збігаються, то вагові коефіцієнти зв'язків не змінюються. В іншому випадку обчислюється різниця між фактичними та необхідними вихідними значеннями, яка передається послідовно від вихідного шару до вхідного. На основі цієї інформації проводиться модифікація зв'язків відповідно до узагальненого дельта-правила, яке має вигляд: $\nabla_p w_{ji} = \eta \delta_{jp} y_{ip}$, де зміна в силі зв'язку w_{ji}

для p -ї навчальної пари $\nabla_p w_{ji}$ пропорційне добутку сигналу помилки j -го нейрона δ_{jp} , який отримує вхідний сигнал по цьому зв'язку, та вихідного сигналу i -го нейрона y_{ip} , який надсилає сигнал по цьому зв'язку. Визначення сигналу помилки є рекурсивним процесом, який починається з вихідних блоків. Для вихідного блоку сигнал помилки $\delta_{jp} = y_j'(T_{jp} - R_{jp})$ де T_{jp}, R_{jp} – це бажане та реальне значення вихідного сигналу j -го блоку, відповідно; y_j' – це похідна від вихідного сигналу j -го блоку. Сигнал помилки для скритого блоку визначається рекурсивно через сигнал помилки блоків, з яким з'єднаний його вихід, та ваги цих зв'язків дорівнюють $\delta_{jp} = y_j' \sum_k \delta_{kp} w_{kj}$. Для сигмоїдної функції похідна дорівнює $y_j' = y_j(1 - y_j)$, тому на інтервалі від 0 до 1 похідна має максимальне значення в точці $y_j = 0.5$, а в точках $y_j = 0$ та $y_j = 1$ звертається в нуль. Максимальна зміна вагів відповідає блокам (нейронам), які ще не обрали свій стан. Крім того, при кінцевих значеннях вагових коефіцієнтів вихідні сигнали блоків не можуть досягати значень 0 або 1. Тому за 0 зазвичай приймають значення $y_j < 0.1$, а за 1 значення $y_j > 0.9$.

Модифікація ваг проводиться після пред'явлення (опрацювання) кожної пари вхід-вихід. Однак якщо коефіцієнт швидкості навчання малий, то можна показати, що узагальнене дельта-правило досить добре апроксимує мінімізацію загальної помилки функціонування мережі D методом градієнтного спуску в просторі ваг:

$$D = \frac{1}{2} \sum_p \sum_j (T_{jp} - R_{jp})^2$$

Навчання триває доти, доки помилка не зменшиться до заданої величини. Емпіричні результати свідчать, що при малих значеннях система знаходить досить хороший мінімум D . Один з основних недоліків алгоритму зворотного поширення помилки полягає в тому, що в багатьох випадках для збіжності може знадобитися багаторазове (сотні разів) пред'явлення (опрацювання) всієї навчальної вибірки. Підвищення швидкості навчання

можна досягти, наприклад, використовуючи інформацію про другу похідну D або шляхом збільшення швидкості навчання.

Алгоритм зворотного поширення помилки використовується також для навчання мереж із зворотними зв'язками. При цьому використовується еквівалентність багат шарової мережі з прямими зв'язками та синхронної мережі зі зворотними зв'язками на обмеженому інтервалі часу (шар відповідає такту часу).

Нині запропоновано алгоритми навчання, привабливіші у сенсі біологічної аналогії. Прикладом є алгоритм рециркуляції для мереж, у яких приховані блоки з'єднані з вхідними. При цьому, при навчанні ваги зв'язків перебудовуються таким чином, щоб мінімізувати частоту зміни активності кожного блоку. Таким чином, навчена мережа має стабільні стани та може функціонувати в режимі асоціативної пам'яті.

Типи завдань машинного навчання та ШНМ

Узагальнимо типи завдань ШНМ. Всі завдання, які вирішуються за допомогою машинного навчання та ШНМ, відносяться до однієї з наступних категорій.

1) *Завдання регресії* – прогноз з урахуванням вибірки об'єктів з різними ознаками. На виході має вийти матеріальне число (2, 35, 76.454 та інші), наприклад вартість квартири, ціна цінного паперу після півроку, очікуваний дохід магазину наступного місяця, якість вина при сліпому тестуванні.

2) *Завдання класифікації* – отримання категоріальної відповіді на основі набору ознак. Має кінцеву кількість відповідей (як правило, у форматі «так» чи «ні»): чи є на фотографії кіт, чи є зображення людським обличчям, чи хворий на рак.

3) *Завдання кластеризації* – розподіл даних на групи: поділ всіх клієнтів мобільного оператора за рівнем платоспроможності, віднесення

космічних об'єктів до тієї чи іншої категорії (планета, зірка, чорна діра тощо).

4) *Завдання зменшення розмірності* – зведення великої кількості ознак до меншого (зазвичай 2–3) для зручності їхньої подальшої візуалізації (наприклад, стиск даних).

5) *Завдання виявлення аномалій* – відокремлення аномалій від стандартних випадків. На перший погляд воно збігається із завданням класифікації, але є одна істотна відмінність: *аномалії* – явище рідкісне, і навчальних прикладів, на яких можна навчити модель на виявлення таких об'єктів або дуже мало, або просто немає, тому методи класифікації тут не працюють. На практиці таким завданням є, наприклад, виявлення шахрайських дій із банківськими картками.

Метрики якості. Функції втрат у завдання регресії.

Усі алгоритми машинного навчання ґрунтуються на мінімізації або максимізації функції, яку ми називаємо *«цільовою функцією»*. Група функцій, що мінімізуються, називається *«функціями втрат»*. *Функція втрат* – це міра того, наскільки добра модель прогнозування з погляду можливості передбачити очікуваний результат. Найчастіше використовуваний метод пошуку точки мінімуму функції – це градієнтний спуск. Насправді, можна уявити функцію втрат, як горбисту гору, а градієнтний спуск – це як ковзання з гори до нижньої точки [14-18].

Немає єдиної функції втрат, яка б працювала із всіма типами даних. Це залежить від ряду чинників, включаючи наявність викидів, вибір алгоритму машинного навчання або архітектури ШНМ, тимчасову ефективність градієнтного спуску, простоту пошуку похідних і надійність прогнозів.

Функції втрат можна умовно поділити на 2 типи, що відповідають розгляненим задачам, що вирішуються за допомогою ШНМ: втрати у

задачах класифікації та регресійні втрати. Давайте розглянемо регресійні втрати.

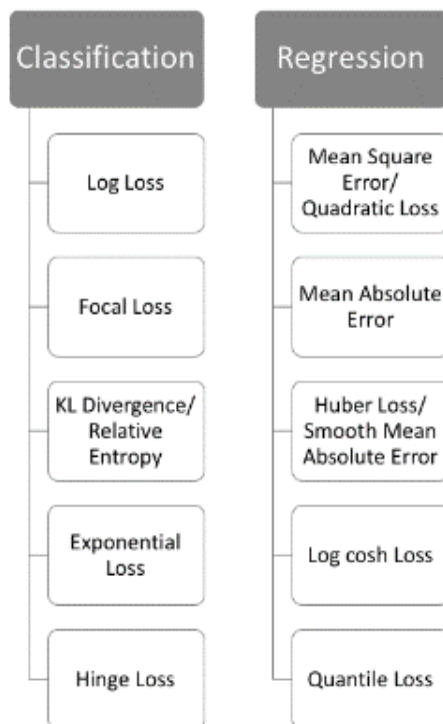


Рисунок 2.2 – Приклади функцій втрат

Функції втрат для задач регресії обчислюють кількість, а функції втрат для задач класифікації обчислюють мітку.

1. Середньоквадратична помилка, квадратичні втрати, втрати L2

Середньоквадратична помилка (mean squared error, MSE) – це функція втрат задач регресії, що найчастіше використовується. MSE – це сума квадратів відстаней між нашою цільовою змінною та прогнозованими значеннями [2-5]:

$$MSE = \frac{\sum_{i=1}^n (y_i - y_i^p)^2}{n}$$

На рисунку 2.3 наведено графік функції MSE, де справжнє цільове значення дорівнює 100, а прогнозовані значення перебувають у діапазоні від

-10 000 до 10 000. Втрати MSE (вісь Y) досягають мінімального значення при прогнозі (вісь X) = 100. Діапазон зміни значень від 0 до ∞ .

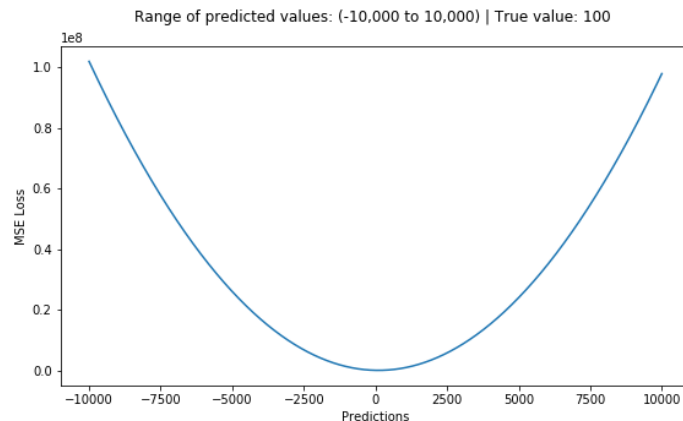


Рисунок 2.3 – Втрати MSE (вісь Y) порівняно з прогнозами (вісь X)

2. Середня абсолютна помилка, втрата L1

Середня абсолютна помилка (MAE) – ще одна функція втрат, яка використовується для регресійних моделей. MAE – це сума абсолютних різниць між нашими цільовими та прогнозованими змінними. Таким чином, він вимірює середню величину помилок у наборі прогнозів, не враховуючи їхнього напрямку. (Якщо ми також розглянемо напрямки, це буде називатися середньою помилкою зміщення, яка є сумою залишків / помилок). Діапазон зміни також від 0 до ∞ .

$$MAE = \frac{\sum_{i=1}^n |y_i - y_i^p|}{n}$$

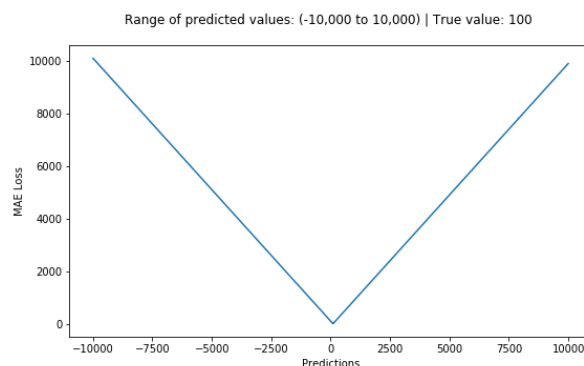


Рисунок 2.4 – Втрати MAE (вісь Y) порівняно з прогнозами (вісь X)

MSE проти MAE (втрата L2 проти втрати L1).

Ці види втрат мають свої плюси і недоліки, зокрема використання абсолютної помилки стійкіше до викидів. Але давайте розберемося чому.

Щоразу, коли ми навчаємо модель машинного навчання, наша мета – знайти точку, яка мінімізує функцію втрат. Звичайно, обидві функції досягають мінімуму, коли прогноз точно дорівнює істинному значенню.

Давайте подивимося значення MAE і середньоквадратичної помилки (RMSE, яка являє собою просто квадратний корінь з MSE, щоб він був у тому ж масштабі, що і MAE) для 2 випадків. У першому випадку прогнози близькі до дійсних значень, і помилка має невелику варіацію між спостереженнями. У другому випадку спостерігається одне відхилення і помилка висока (рисунок 2.5).

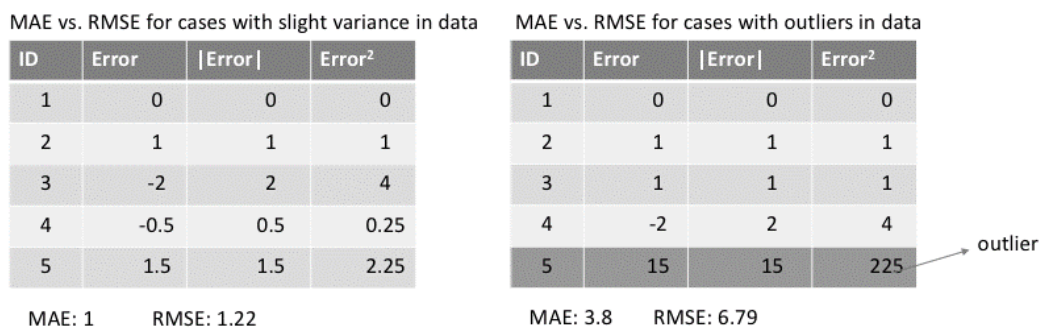


Рисунок 2.5 – Приклад роботи MAE та RMSE на різних даних

На рисунку 2.5 зліва – помилки близькі одна до одної, а праворуч: одна помилка відрізняється від інших.

Що ми з цього спостерігаємо і як це допоможе вибрати, яку функцію втрат використовувати?

Оскільки MSE зводить помилку у квадрат ($y_i - y_i^p = e$), значення помилки (e) значно збільшується, якщо $e > 1$. Якщо ми маємо викид у наших даних, значення e буде більшим, а e^2 буде $\gg |e|$. Це змусить модель із втратами MSE надати більшу вагу викидам, ніж модель із втратами MAE. У другому наведеному вище випадку модель з RMSE втратами буде

скоригована, щоб мінімізувати цей єдиний випадок викиду за рахунок інших поширених прикладів, що знизить її загальну продуктивність.

Отже, втрата MAE корисна, якщо дані навчання пошкоджені викидами (тобто ми помилково отримуємо нереально величезні негативні/позитивні значення у нашому середовищі навчання, але не в середовищі тестування). Інтуїтивно ми можемо думати про це так: якби нам потрібно було дати лише один прогноз для всіх спостережень, які намагаються мінімізувати MSE, то цей прогноз має бути середнім для всіх цільових значень. Але якщо ми спробуємо мінімізувати MAE, це прогноз буде медіанним для всіх спостережень. Ми знаємо, що медіана стійкіша до викидів, ніж середнє значення, що, отже, робить MAE більш стійким до викидів, ніж MSE.

Одна велика проблема при використанні втрат MAE (особливо для нейронних мереж) полягає в тому, що його градієнт є однаковим, що означає, що градієнт буде великим навіть для невеликих значень втрат. Це погано для навчання. Щоб виправити це, ми можемо використовувати динамічну швидкість навчання, яка зменшується в міру наближення до мінімумів. MSE у цьому випадку поводить себе добре і сходиться навіть при фіксованій швидкості навчання. Градієнт втрат MSE високий для великих значень втрат і зменшується з наближенням втрат до 0, що робить його більш точним наприкінці навчання (рисунок 2.6).

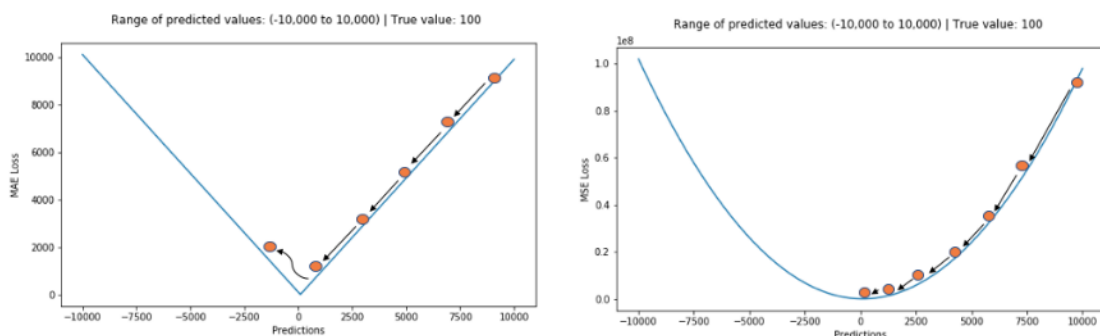


Рисунок 2.6 – навчання із MAE (зліва) та MSE (праворуч) втратами.

Як же вирішити, яку функцію втрат використовувати? Якщо викиди є аномаліями, які важливі для бізнесу і повинні бути виявлені, ми повинні використовувати MSE. З іншого боку, якщо ми вважаємо, що викиди є просто пошкодженими даними, тоді ми повинні вибрати MAE як втрату.

Втрати L1 більш стійкі до викидів, але їх похідні є безперервними, що робить пошук рішення неефективним. Втрати L2 чутливі до викидів, але дають більш стабільне і закрите рішення (встановивши його похідну = 0).

Проблеми з обома функціями: можуть бути випадки, коли жодна функція втрат не дає бажаних прогнозів. Наприклад, якщо 90% спостережень у наших даних мають справжнє цільове значення 150, а решта 10% мають цільове значення від 0 до 30. Тоді модель з MAE втратами може прогнозувати 150 для всіх спостережень, ігноруючи 10% викидів, оскільки вона намагатиметься наблизитись до середнього значення. В тому ж випадку модель, що використовує MSE втрати, дасть багато прогнозів у діапазоні від 0 до 30, оскільки вона буде зміщена у бік викидів. Обидва результати небажані у багатьох бізнес-випадках.

Що робити у такому разі? Найпростішим рішенням буде перетворення цільових змінних. Інший спосіб – спробувати іншу функцію втрат.

3. Втрата Хубера (Huber loss), гладка середня абсолютна помилка.

Втрати Хубера менш чутливі до викидів у даних, ніж втрати MSE. Вони також диференціюються при 0. Втрати Хубера являють собою переважно абсолютну помилку (MAE), які стають квадратичними (MSE), коли помилка мала. Наскільки мала ця помилка, щоб зробити її квадратичною, залежить від гіперпараметра δ , який можна налаштувати. Втрати Хубера наближаються до MSE, коли $\delta \sim 0$, і MAE, коли $\delta \sim \infty$ (великі числа).

$$L_{\delta}(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{for } |y - f(x)| \leq \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2, & \text{otherwise.} \end{cases}$$

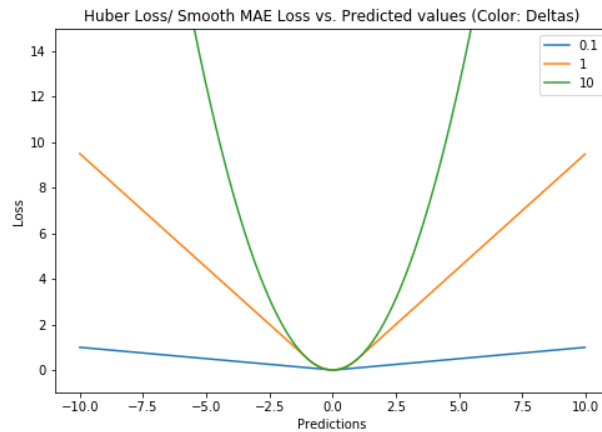


Рисунок 2.7 – втрати Хубера (вісь Y) порівняно з прогнозами (вісь X).

Справжнє значення = 0.

Вибір δ має вирішальне значення, тому що воно визначає, що ви хочете розглядати як викид. Залишки, що більші за δ мінімізуються за допомогою L1 (який менш чутливий до викидів), а залишки менші за δ мінімізуються «належним чином» за допомогою L2.

Для чого використовувати Huber Loss? Одна велика проблема з використанням MAE для навчання нейронних мереж – це великий градієнт, який може призвести до відсутності мінімумів наприкінці навчання з використанням градієнтного спуску. Для MSE градієнт зменшується з наближенням втрат до мінімуму, що робить його більш точним.

У таких випадках можуть бути дуже корисними втрати Хубера, оскільки вони згинаються навколо мінімумів, що зменшує градієнт. І ці втрати стійкіший до викидів, ніж MSE. Отже, втрати Хубера поєднують у собі хороші властивості як MSE так і MAE. Однак проблема із втратами Хубера полягає в тому, що нам може знадобитися навчити параметр δ як гіперпараметр, що є ітеративним процесом.

4. Логарифмічні втрати

Log-ch – це ще одна функція втрат, що використовується в задачах регресії, більш плавна, ніж L2. Log-ch – це логарифм гіперболічного косинуса помилки передбачення:

$$L(y, y^p) = \sum_{i=1}^n \log(\cosh(y_i^p - y_i))$$

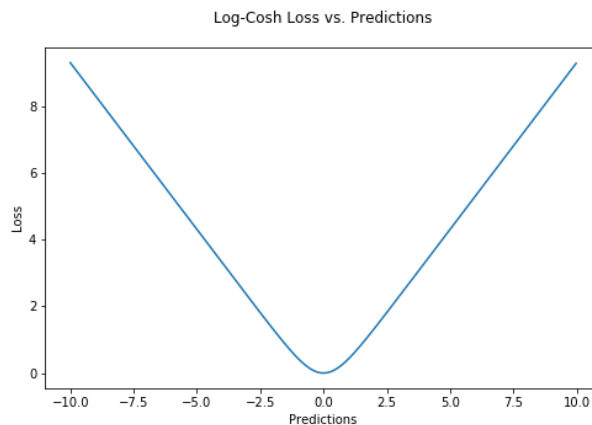


Рисунок 2.8 – Графік логарифмічних втрат (вісь Y) залежно від прогнозів (вісь X). Справжнє значення = 0

Перевага: $\log(\cosh(x))$ приблизно дорівнює $(x^2) / 2$ для малих x та $\log(2)$ для великих x . Це означає, що $\logcosh$ працює в основному як середньоквадратична помилка, але на нього не так сильно впливають випадкові абсолютно невірні прогнози. Ці втрати мають всі переваги втрат Хубера, і скрізь двічі диференційовані, на відміну від втрати Хубера.

Навіщо потрібна друга похідна? Багато реалізацій моделей машинного навчання, такі як **XGBoost**, використовують метод Ньютона для пошуку оптимуму, тому необхідна друга похідна (Гесіан). Тому, для фреймворків машинного навчання, таких як **XGBoost**, краще мати двічі диференційовані функції втрат.

Але логарифмічні втрати не є ідеальними. Ці втрати все ще страждають від проблеми, пов'язаної з сталістю градієнта і гесіана для дуже

великих нецільових прогнозів, що призводить до відсутності поділів для **XGBoost**.

5. Квантильні втрати

У більшості завдань прогнозування у реальному світі нам часто цікаво дізнатися про невизначеність наших прогнозів. Знання діапазону прогнозів, а не лише точкових оцінок може значно покращити процеси прийняття рішень для багатьох бізнес-проблем.

Квантильні функції втрат є корисними, коли нас цікавить прогнозування інтервалу, а не тільки точкове прогнозування. Інтервал прогнозу на основі регресії найменших квадратів ґрунтується на припущенні, що залишки $(y_i - y_i^p)$ мають постійну дисперсію за значеннями незалежних змінних. Ми не можемо довіряти моделям лінійної регресії, які порушують це припущення. Ми також не можемо просто відкинути ідею підбору моделі лінійної регресії як основу, заявивши, що такі ситуації завжди краще моделювати з використанням нелінійних функцій чи моделей на основі дерев. Тут на допомогу приходять квантильні втрати та квантильна регресія, оскільки регресія, що базується на квантильних втратах, забезпечує розумні інтервали прогнозування навіть для залишків із непостійною дисперсією чи ненормальним розподілом.

Поглянемо на робочий приклад, щоб краще зрозуміти, чому регресія на основі квантильних втрат добре працює з гетероскедастичними даними (рисунок 2.9, 2.10, 2.11).

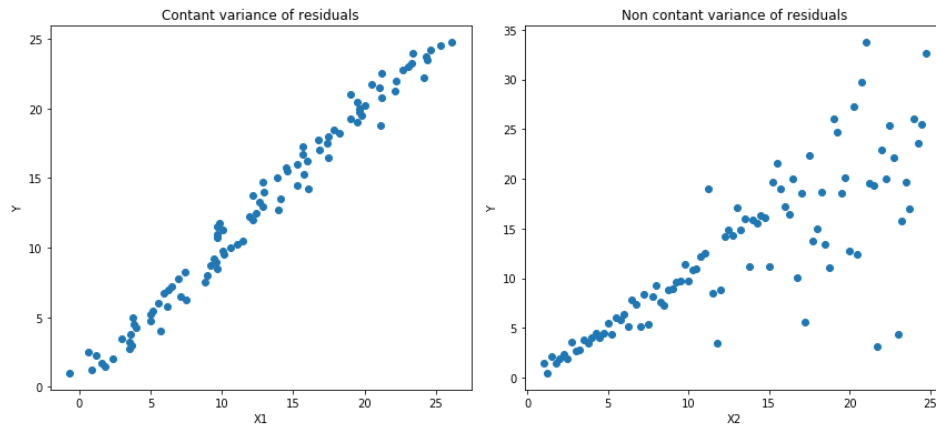


Рисунок 2.9 – Приклад гетероскедастичності. Зліва: лінійна залежність X_1 та Y . З постійною дисперсією залишків; праворуч: лінійні відносини X_2 та Y , але відхилення Y збільшується з X_2 (Гетероскедастичність).

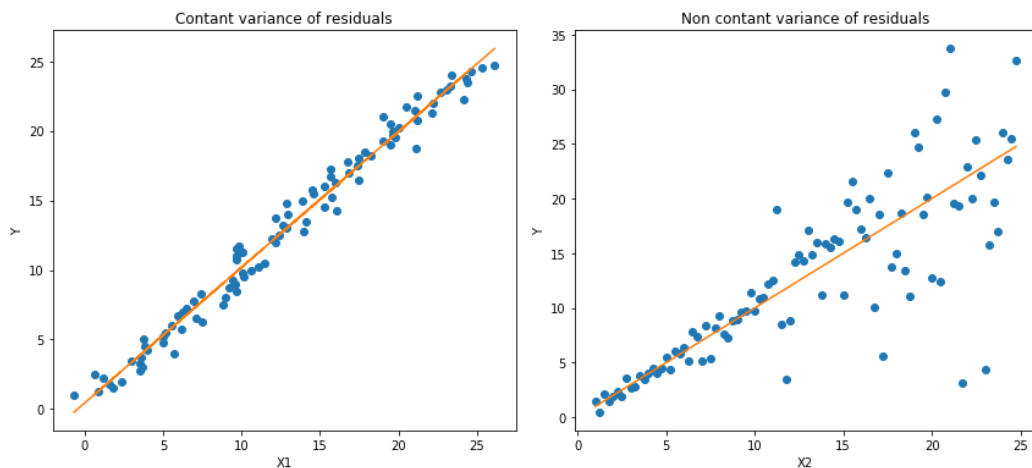


Рисунок 2.10 – Звичайна регресія методом найменших квадратів.

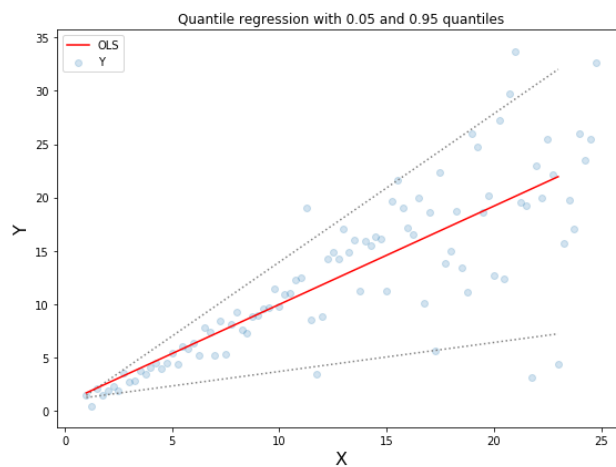


Рисунок 2.11 – Квантильна регресія. Пунктирні лінії представляють функції втрат 0,05 та 0,95 (квантилі) на основі квантильної регресії.

Розуміння функції квантильних втрат

Регресія на основі квантилів спрямована на оцінку умовного «квантилю» змінної відгуку за певних значень змінних-предикторів. Квантильні втрати є насправді просто продовженням MAE (коли квантиль становить 50-й процентиль, це MAE).

Ідея полягає в тому, щоб вибрати значення квантилю в залежності від того, чи хочемо ми надати більшого значення позитивним або негативним помилкам. Функція втрат намагається дати різні штрафи за переоцінку та недооцінку на основі значення вибраного квантилю (γ). Наприклад, функція квантильних втрат $\gamma = 0,25$ дає більший штраф за переоцінку і намагається зберегти значення прогнозу трохи нижче за медіану.

$$L_{\gamma}(y, y^p) = \sum_{i=y_i < y_i^p} (\gamma - 1) \cdot |y_i - y_i^p| + \sum_{i=y_i \geq y_i^p} (\gamma) \cdot |y_i - y_i^p|$$

де γ – це потрібний квантиль, що має значення від 0 до 1.

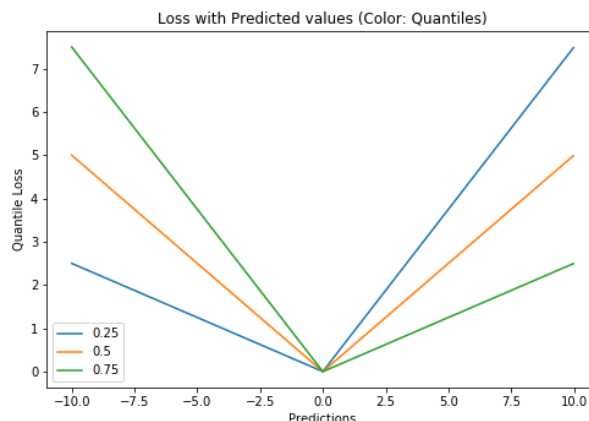


Рисунок 2.12 – Графік квантильних втрат (вісь Y) порівняно з прогнозами (вісь X). Справжнє значення $Y = 0$

Ми також можемо використовувати цю функцію втрат для розрахунку інтервалів прогнозування у нейронних мережах або моделях на основі дерев.

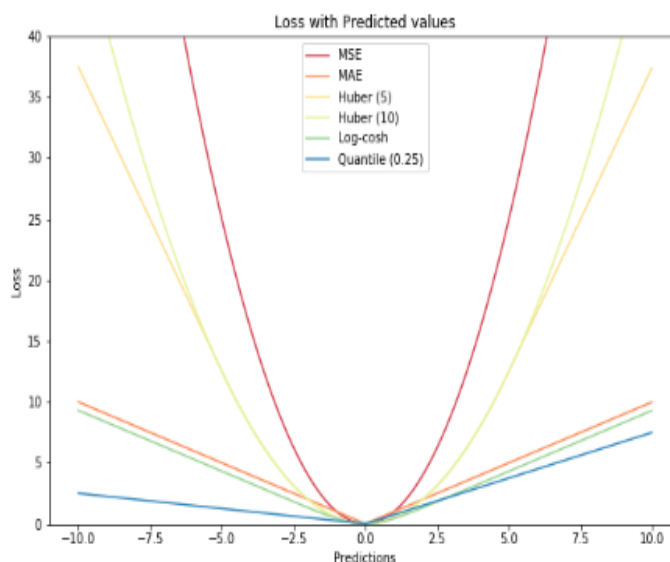


Рисунок 2.13 – Регресійні функції втрат

Деякі практичні спостереження:

- Прогнози моделі з втратами MAE менш схильні до впливу імпульсного шуму, тоді як прогнози з функцією втрат MSE злегка зміщені через викликані відхилення.
- Прогнози мало чутливі до значення гіперпараметра, обраного у разі використання моделі із втратою Huber.
- Квантильні втрати дають відмінну оцінку відповідних рівнів достовірності.

Загальні обмеження на функцію втрат (ФВ).

- ФВ має бути диференційована, тобто ми за будь-яких параметрів повинні вміти обчислити похідну, але якщо в якійсь точці похідна невизначена (наприклад, у точці 0) – можна її довизначити.
- Похідна ФВ в якийсь досить великій множині повинна бути відмінна від 0, у протилежному випадку ФВ буде неінформативна і не буде говорити в яку сторону покращувати параметри.

Глибокі нейронні мережі.

Глибоке навчання (deep learning) – це клас алгоритмів машинного навчання, які навчаються глибше (абстрактніше) розуміти дані. Популярні архітектури нейронних мереж глибокого навчання представлені на рисунку 2.14.

Розглянемо загальні принципи побудови нейронних мереж.

Розглянемо одношарову нейронну мережу (рисунок 2.15). Тут навчається перший шар (зелені нейрони), він просто передається на вихід.

У той час як у випадку двошарової нейронної мережі, незалежно від того, як навчається прихований зелений шар, він потім передається на синій прихований шар, де продовжує навчатися (рисунок 2.16).

Отже, що більше число прихованих шарів, то більше можливості навчання мережі.

Не слід плутати із *широкою ШНМ*. У цьому випадку велика кількість нейронів в одному шарі не призводить до глибокого розуміння даних. Але це призводить до вивчення більшої кількості ознак (рисунок 2.17) [4-8, 12-18].

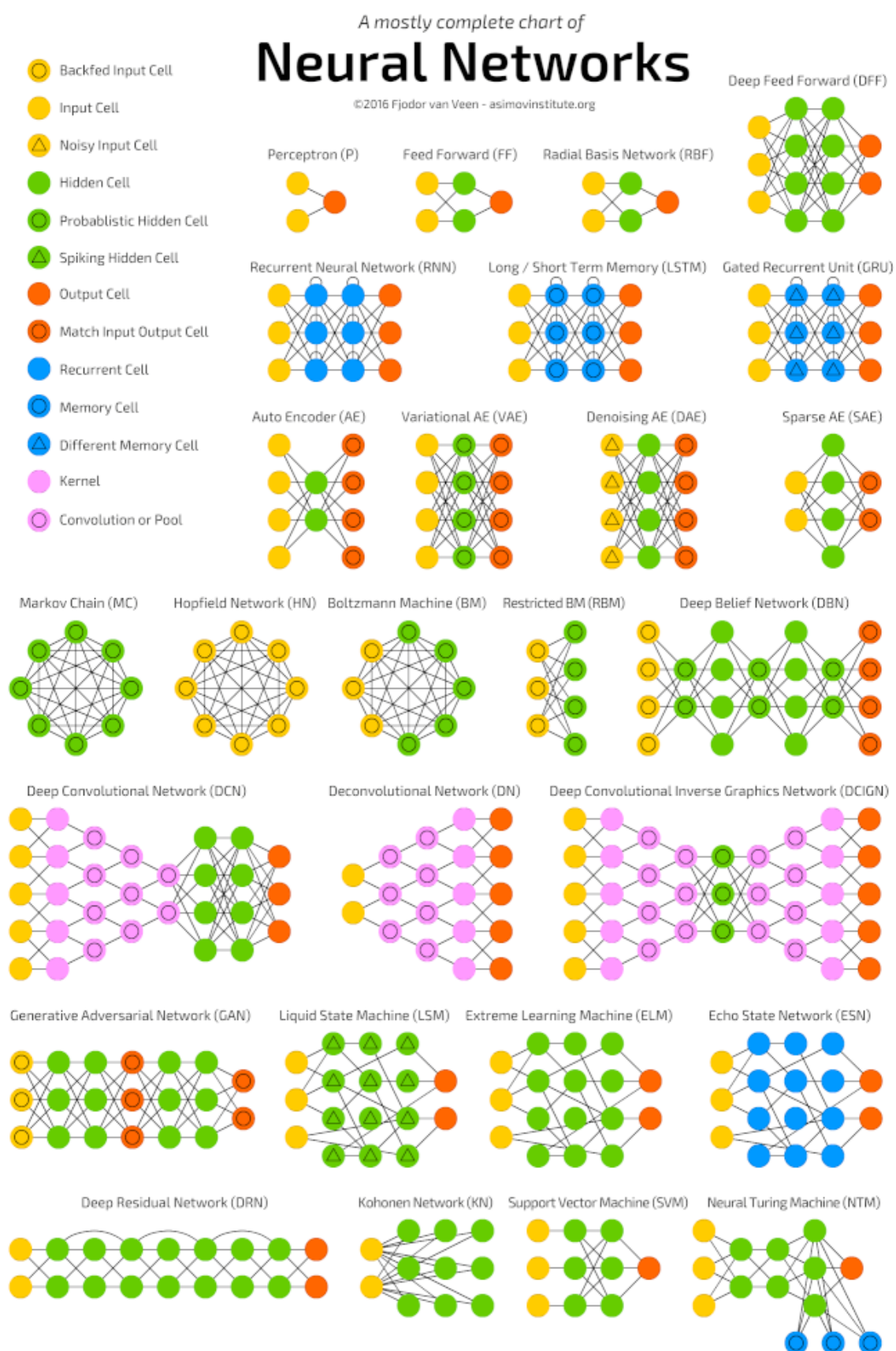


Рисунок 2.14 – Популярні архітектури нейронних мереж
(<http://www.asimovinstitute.org/neural-network-zoo>)

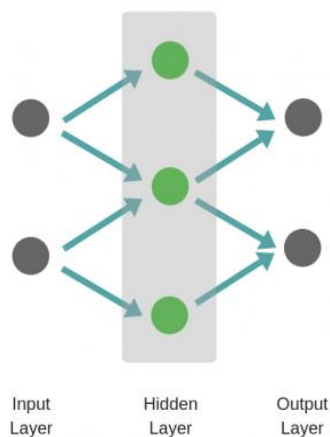


Рисунок 2.15 – Приклад одношарової нейронної мережі

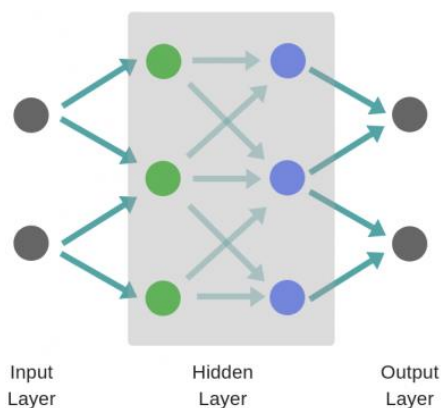


Рисунок 2.16 – Приклад двошарової нейронної мережі

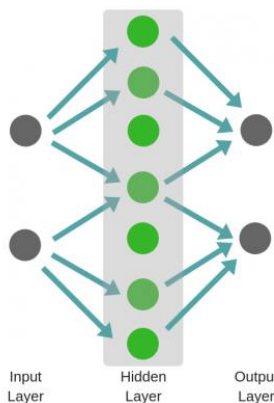


Рисунок 2.17 – Приклад одношарової широкої нейронної мережі

Приклад: Вивчаючи англійську граматику, потрібно знати безліч понять. У цьому випадку одношарова широка нейронна мережа працює набагато краще, ніж глибока нейронна мережа, яка значно менша. Але, у разі вивчення перетворення Фур'є, учень (нейронна мережа) має бути глибоким, тому що треба вивчити не так багато понять, які потрібно знати,

але кожне з них є досить складним і потребує глибокого розуміння. Головне – це баланс.

Дуже привабливо використовувати глибокі та широкі нейронні мережі для кожного завдання. Але це може бути поганою ідеєю, тому що:

- Обидві вимагають значно більшої кількості даних для навчання, щоб досягти мінімальної бажаної точності;
- Обидві мають експоненційну складність;
- Занадто глибока нейронна мережа спробує зламати фундаментальні уявлення, але при цьому вона робитиме помилкові припущення і намагатиметься знайти псевдозалежності, які не існують;
- Занадто широка нейронна мережа намагатиметься знайти більше ознак, ніж є. Таким чином, подібно до попередньої, вона почне робити неправильні припущення про дані.

Прокляття розмірності. Недонавчання та перенавчання.

Прокляття розмірності відноситься до різних явищ, що виникають при аналізі та організації даних у багатовимірних просторах (часто з сотнями або тисячами вимірів), і не зустрічається в ситуаціях з низькою розмірністю.

У машинному навчанні ми повинні представити дані ознаками у вигляді масиву/матриці кінцевої та суттєво меншої довжини (ніж кількість існуючих ознак). Для цього нейронні мережі узагальнюють ці ознаки. Це породжує дві проблеми:

- Через неправильні припущення з'являється зміщення. Високі значення зміщення можуть призвести до того, що алгоритм пропустить суттєвий взаємозв'язок між ознаками та цільовими змінними. Це явище називають **недонавчання**.
- Від невеликих відхилень у навчальній множині через недостатнє вивчення ознак збільшується дисперсія. Висока дисперсія

веде до **перенавчання**, коли помилки сприймаються як надійна інформація.

Компроміс

На ранній стадії навчання зміщення велике, тому що вихід із мережі далекий від бажаного. А дисперсія дуже мала, оскільки дані мають поки що малий вплив.

Наприкінці навчання зміщення невелике, тому що мережа виявила основну функцію в даних. Однак, якщо навчання є занадто тривалим, мережа також вивчить шум, характерний для цього набору даних. Це призводить до великого розкиду результатів при тестуванні на різних множинах, оскільки шум змінюється від набору даних до іншого.

Справді, алгоритми з **великим зміщенням** зазвичай в основі простіших моделей, які не схильні до перенавчання, але можуть недонавчитися і не виявити важливі закономірності або властивості ознак. Моделі з **невеликим зміщенням** і великою дисперсією зазвичай складніші з погляду їх структури, що дозволяє більш точно представляти навчальний набір. Однак вони можуть відображати багато шуму з навчального набору, що робить їх прогнози менш точними, незважаючи на їхню додаткову складність.

Отже, як правило, неможливо мати маленьке зміщення і маленьку дисперсію одночасно.

Наразі є безліч інструментів, за допомогою яких можна легко створити складні моделі машинного навчання, перенавчання займає центральне місце. Зміщення з'являється, коли мережа не отримує достатньо інформації. Але чим більше прикладів, тим більше з'являється варіантів залежностей та змінностей у цих кореляціях.

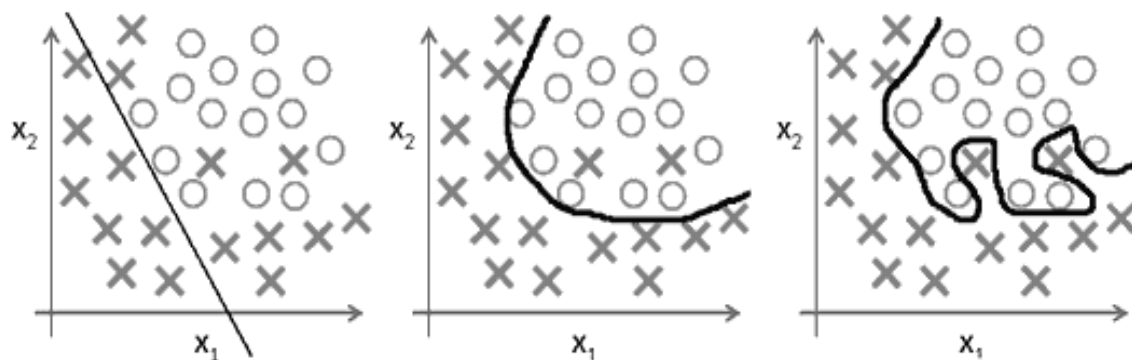


Рисунок 2.18 – Приклад недонавчання (зліва), збалансованої моделі (в центрі) та перенавчання (праворуч)

Перенавчання, як випливає з назви, це стан нейромережі, коли вона перенасичена даними. Ця проблема виникає, якщо занадто довго навчати мережу на тих самих даних. Іншими словами, мережа почне не вчитися на даних, а запам'ятовувати і зубрити їх. Відповідно, коли ви вже подаватимете на вхід цієї мережі нові дані, то в отриманих даних може з'явитися шум, який впливатиме на точність результату. Наприклад, якщо ми будемо показувати мережі різні фотографії яблук (тільки червоні) і говорити, що це яблуко. Тоді, коли НМ побачить жовте або зелене яблуко, воно не зможе визначити, що це яблуко, оскільки вона запам'ятала, що всі яблука мають бути червоними. І навпаки, коли НМ побачить щось червоне і за формою, що збігається з яблуком, наприклад персик, вона скаже, що це яблуко. Це і є шум. На рисунку 2.19 наведено приклад даних із шумом.

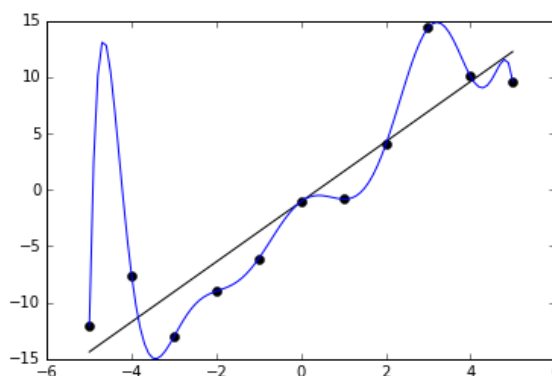


Рисунок 2.19 – Приклад даних із шумом

Видно, що графік функції сильно коливається від точки до точки, які є вихідними даними (результатом) нашої мережі. В ідеалі цей графік повинен бути менш хвилястий і прямий. Щоб уникнути перенавчання, не варто довго тренувати нейромережу на одних і тих самих або дуже схожих даних. Також, перенавчання може бути викликане великою кількістю параметрів, які ви подаєте на вхід ШНМ або надто складною архітектурою. Таким чином, коли ви помічаєте помилки (шум) у вихідних даних після етапу навчання, вам варто використовувати один з методів **регуляризації**, але в більшості випадків це не знадобиться.

Як зробити, щоб ШНМ давала правильні відповіді? Градієнтний спуск, зворотне поширення помилки.

Відповідь проста – потрібно її навчати. Однак, наскільки б простою не була відповідь, її реалізація в плані простоти залишає бажати кращого. Існує кілька методів навчання ШНМ:

- Метод зворотного розповсюдження (МЗР, Backpropagation)
- Метод пружного розповсюдження (Resilient propagation або Rprop)
- Генетичний Алгоритм (ГА, Genetic Algorithm)

Розглянемо більш детально метод зворотного розповсюдження помилки, який використовує алгоритм градієнтного спуску, та є найбільш вживаним у останні роки для тренування ШНМ (Rprop і ГА виходять за рамки даного навчального посібнику).

Що таке градієнтний спуск?

Це спосіб знаходження локального мінімуму або максимуму функції за допомогою руху вздовж градієнта. Якщо ви зрозумієте суть градієнтного спуску, то у вас не повинно виникнути жодних питань під час використання методу зворотного розповсюдження помилки. Для початку, давайте розберемося, що таке градієнт і де він присутній у нашій ШНМ. Давайте

побудуємо графік, де по осі x будуть значення ваги нейрона(w), а по осі y — помилка, що відповідає цій вазі(e) (рисунок 2.20) [4-8, 12-14].

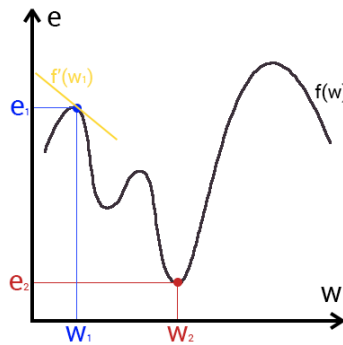


Рисунок 2.20 – Приклад визначення градієнта функції

Подивившись на графік на рисунку 2.20, ми зрозуміємо, що графік функція $f(w)$ є залежністю помилки від вибраної ваги. На цьому графіку нас цікавить *глобальний мінімум* – точка (w_2, e_2) або, іншими словами, те місце, де графік підходить ближче до осі x . Ця точка означатиме, що вибравши вагу w_2 ми отримаємо найменшу помилку – e_2 і, як наслідок, найкращий результат із усіх можливих. Знайти цю точку нам допоможе метод градієнтного спуску (жовтим на графіку позначений градієнт). Відповідно, у кожної ваги в нейромережі буде свій графік і градієнт і у кожного треба знайти глобальний мінімум.

То що таке, цей градієнт? Градієнт – це вектор, який визначає крутість схилу і вказує його напрямок щодо будь-якої з точок на поверхні. Щоб знайти градієнт потрібно взяти похідну від графіка за цією точкою. Рухаючись у напрямку цього градієнта, ми будемо плавно скочуватися в низину (рисунок 2.21). Тепер уявімо, що помилка – це лижник, а графік функції – це гора. Відповідно, якщо помилка дорівнює 100%, то лижник перебуватиме на самій вершині гори і якщо помилка 0% то в низині. Як усі лижники, помилка прагне якнайшвидше спуститися вниз і зменшити своє значення. У кінцевому випадку у нас має вийти результат, як позначено на рисунку 2.21.

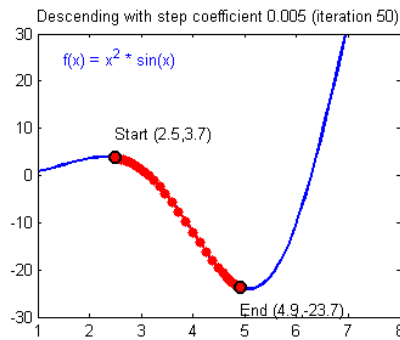


Рисунок 2.21 – Приклад розрахунку градієнта функції

Уявіть, що лижника закидають, за допомогою гелікоптера, на гору. Наскільки високо чи низько залежить від випадку (аналогічно до того, як у нейронній мережі, при ініціалізації, ваги розставляються у випадковому порядку). Припустимо, помилка дорівнює 90% і це наша точка відліку. Тепер лижнику потрібно спуститися вниз за допомогою градієнта. На шляху вниз, у кожній точці ми будемо обчислювати градієнт, що буде показувати напрям спуску і при зміні нахилу, коригувати його. Якщо схил буде прямим, то після n -ої кількості таких дій ми дістанемося до низини. Але здебільшого схил (графік функції) буде хвилястий і наш лижник зіткнеться з дуже серйозною проблемою – *локальний мінімум*.

Потрапляння в локальний мінімум загрожує тим, що наш лижник назавжди залишиться в цій низині і ніколи не скотиться з гори, тому ми ніколи не зможемо отримати правильну відповідь. Але ми можемо уникнути цього, спорядивши нашого лижника реактивним ранцем під назвою момент (momentum). Коротка ілюстрація моменту наведена на рисунку 2.22.



Рисунок 2.22 – Дія моменту (інерції) під час градієнтного спуску

Як ви вже напевно здогадалися, цей ранець додає лижнику необхідне прискорення, щоб подолати пагорб, який утримує нас у локальному мінімумі, проте тут є одне *АЛЕ*. Уявімо, що ми встановили певне значення параметра *момент* і легко змогли подолати всі локальні мінімуми, і дістатися до глобального мінімуму. Так як ми не можемо просто відключити реактивний ранець, ми можемо проскочити глобальний мінімум, якщо поруч із ним є ще низини. В кінцевому випадку це не так важливо, тому що рано чи пізно ми все одно повернемося назад у глобальний мінімум, але варто пам'ятати, що чим більше момент, тим більше буде розмах з яким лижник кататиметься по низинах. Разом з моментом у методі зворотного поширення помилки також використовується такий параметр, як швидкість навчання (learning rate). Як напевно багато хто подумає, чим більше швидкість навчання, тим швидше ми навчимо нейромережу. Ні. Швидкість навчання, як і момент, є гіперпараметром – тобто величина яка підбирається шляхом спроб і помилок. Швидкість навчання можна безпосередньо пов'язати зі швидкістю лижника і можна з упевненістю сказати – тихіше їдеш далі будеш. Однак тут теж є певні аспекти, тому що якщо ми зовсім не дамо лижнику швидкості, то він взагалі нікуди не поїде, а якщо дамо маленьку швидкість то час шляху може розтягнутися на дуже великий період часу. Що ж тоді станеться, якщо ми дамо занадто велику швидкість?

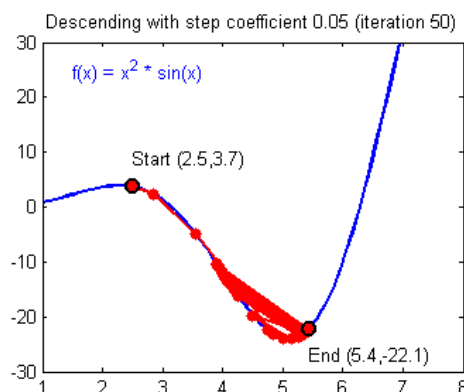


Рисунок 2.23 – Приклад градієнтного спуску з великою швидкістю

Як бачите, нічого хорошого. Лижник почне скочуватися неправильним шляхом і можливо навіть в іншому напрямку, що, як ви розумієте, тільки віддалить нас від знаходження правильної відповіді. Тому у всіх цих параметрах потрібно знаходити золоту середину, щоб уникнути розбіжності нейронної мережі під час навчання.

Часткові похідні.

Часткові похідні можна обчислити, тому відомо, який був внесок у помилку за кожною вагою. Необхідність похідних очевидна. Уявіть нейронну мережу, яка намагається знайти оптимальну швидкість безпілотного автомобіля. Якщо машина виявить, що вона їде швидше або повільніше за потрібну швидкість, нейронна мережа змінюватиме швидкість, прискорюючи або сповільнюючи автомобіль. Що при цьому прискорюється/сповільнюється? Похідні швидкості [3-5].

Розберемо необхідність часткових похідних на простому прикладі. Припустимо, дітей попросили кинути дротик у ціль, цілячись у центр (результат цього процесу зображений на рисунку 2.24).



Рисунок 2.24 – Результат кидання дротика у ціль

Тепер, якщо ми знайдемо загальну помилку і просто віднімемо її з усіх ваг, ми узагальнимо помилки, допущені кожним. Отже, скажімо, дитина потрапила надто низько, але ми просимо всіх дітей прагнути потрапляти в ціль (рисунок 2.25):

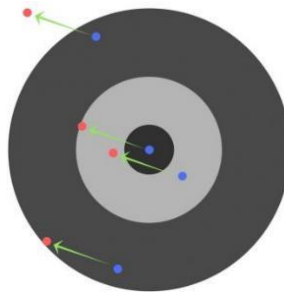


Рисунок 2.25 – Результат після першого корегування

Помилка кількох дітей може зменшитись, але загальна помилка все ще збільшується. Знайшовши часткові похідні, ми дізнаємося про помилки, що відповідають кожній вазі окремо. Таким чином, якщо вибірково виправити ваги, можна отримати результат зображений на рисунку 2.26.

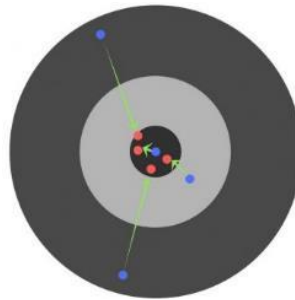


Рисунок 2.26 – Результат після наступного корегування

Поняття графа обчислень. Підрахунок складних градієнтів. Метод ланцюжків.

Розглянемо наступну класифікаційну модель. Функція f приймає дані x та параметри W на вхід та видає вектор оцінок s для кожної з категорій, які потрібно класифікувати. Також у нас є функція втрат L (наприклад, SVM – це hinge функція втрат для задач класифікації, яку ми будемо розглядати у наступному розділі, наразі це не принципово), що визначає наскільки правильні вийшли оцінки. За її допомогою ми можемо обчислити втрати. Нехай, також, буде додаватися регуляризація, детальніше про яку буде йти мова у наступних розділах.

Наша мета – знайти параметри W , що відповідають найменшим втратам. Будемо використовувати градієнтний спуск для навчання нашої

моделі, тобто ми використовуємо негативний напрям градієнта функції L , який вказує шлях до її мінімуму.

$$s = f(x; W) = Wx - \text{Функція оцінки}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) - \text{SVM втрати}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \sum_k W_k^2 \text{ Втрати та член регуляризації}$$

$$\nabla_W L - ? - \text{Шукаємо}$$

Є два основні способи обчислити градієнт: аналітичний та числовий. Числовий метод досить простий, але працює дуже повільно та видає приблизні значення. Аналітичний градієнт більш точний і швидкий, але при його підрахунку можна легко припуститися помилок. Щоб уникнути цього і легко обчислювати градієнт навіть для складних функцій, краще використовувати **обчислювальні графи**.

Обчислювальний граф – це ілюстрований запис будь-якої функції, що складається з вершин та ребер. Вершини (іноді ще називають вузли) — обчислювальні операції, які потрібно виконати, а ребра пов'язують їх в певну послідовність.

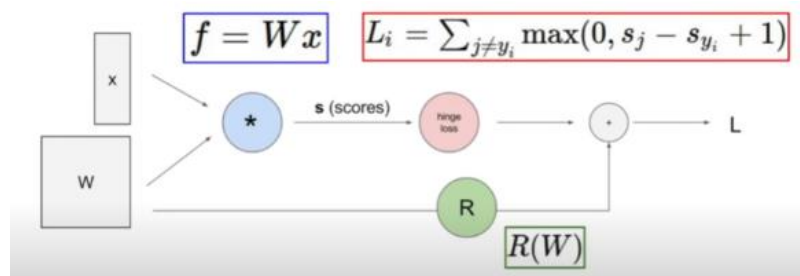


Рисунок 2.27 – Приклад графу обчислень

На рисунку 2.27 наведено приклад графа з нашим класифікатором. Вузол з операцією (*) означає множення матриць параметрів W і даних x результатом якого є вектор ваг s . Наступна вершина залежних втрат (hinge loss) визначає втрати даних L . Вузол R обчислює регуляризацію. І, нарешті, наприкінці ми отримуємо загальні втрати, підсумовуючи регуляризацію та втрати даних.

Перевага графів у тому, що вони дозволяють більш інтуїтивно зрозуміло використовувати метод зворотного поширення помилки (backpropagation). Цей алгоритм рекурсивно використовує правило **диференціювання складної функції** обчислення градієнта кожної змінної у графі. Метод стає дуже корисним для справді складних функцій, що застосовуються у згорткових нейромережах. Розглянемо як він працює на дуже простому прикладі (рисунок 2.28).

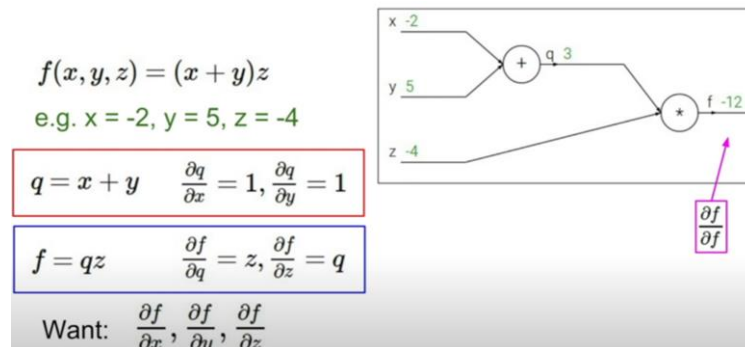


Рисунок 2.28 – Приклад простого графу обчислень та розрахунок зворотного поширення помилки

Ми маємо функцію $f(x, y, z) = (x + y)z$ і ми хочемо знайти її градієнти по відношенню до кожної зі змінних. На обчислювальному графі відображені операції $x + y$ та $(x + y)z$ у вигляді вершин складання (+) та множення (*). Наприклад взято значення $x = -2, y = 5, z = -4$, тому: $-2 + 5 = 3$ (проміжний результат після вузла складання), а $3 * (-4) = -12$ (результат множення).

Позначимо літерами проміжні значення. Нехай змінна після (+) називається q ($q = x + y$), тоді функція f дорівнюватиме qz . Випишемо градієнти (часткові похідні): градієнт q залежить від змінних x і y , а градієнт f – від q і z . Оскільки ми шукаємо градієнт вихідної функції f , яка залежить від усіх трьох змінних x, y та z , то нам хотілося б знайти df/dx , df/dy та df/dz .

Часткові похідні dq/dx і dq/dy дорівнюють одиниці – це означає, що при будь-якій зміні x або y – q зміниться так само.

Спробуємо рекурсивно скористатися правилом диференціювання складної функції (рисунок 2.29). Почнемо з кінця обчислювального графа і рухатимемося до початку, по дорозі рахуючи всі градієнти. Останнє отримане нами значення – функція f і її проміжний результат - 12, часткова похідна за яким дорівнює одиниці. Далі йде змінна z і ми знаємо, що $df/dz = q = 3$. Переходимо до вершини (+) і змінної q : похідна df/dq дорівнює z , тобто - 4.

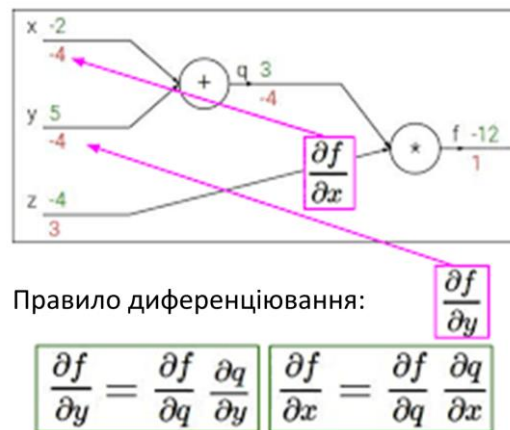


Рисунок 2.29 – Правило диференціювання складної функції

Вийшло досить цікаво: у двох ребрах, що входять у вершину множення, градієнт кожного входу дорівнює значенню іншого входу, помноженому на попередній градієнт. І це не збіг – можете взяти правило на замітку.

Тепер розрахуємо похідну df/dy . Правило диференціювання складної функції говорить, що вона може бути записана як $df/dq * dq/dy$. Значення df/dq та dq/dy нам відомі: вони дорівнюють - 4 і 1 відповідно, тому похідна df/dy дорівнює - 4. Так само це працює і для df/dx .

Виявляється ще одна закономірність: у вершині додавання оказались рівними всі три градієнти: і самої вершини, і вершин, що входять до неї.

Що таке Метод зворотного поширення (МЗП)?

Нарешті, зібравши наведену вище інформацію воєдино, проаналізуємо, як же все-таки зробити так, щоб нейронна мережа могла

правильно навчатися і давати вірні рішення. Для ілюстрації цього процесу будемо використовувати нейронну мережу що складається з трьох шарів (не враховується «2 нейрони» вхідного шару) і має два входи і один вихід (рисунок 2.30).

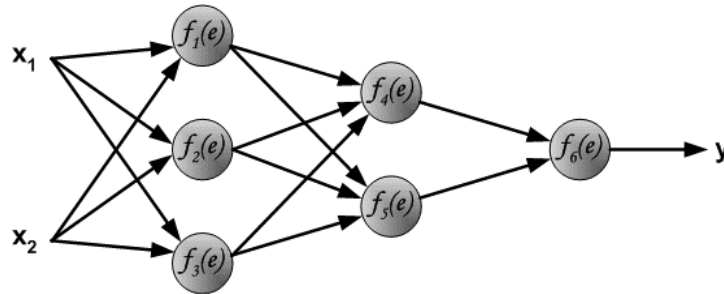


Рисунок 2.30 – Двошарова нейронна мережа для аналізу МЗП

Як ми вже знаємо, кожен нейрон складається з двох елементів (рисунок 2.31).

Перший елемент – дендрити – додають вагові коефіцієнти до вхідних сигналів.

Другий елемент – тіло нейрона – реалізує нелінійну функцію, так звану функцію активації нейрона. Які існують функції активації ми вже знаємо.

Сигнал e – це зважена сума вхідних сигналів

$y = f(e)$ – вихідний сигнал нейрона

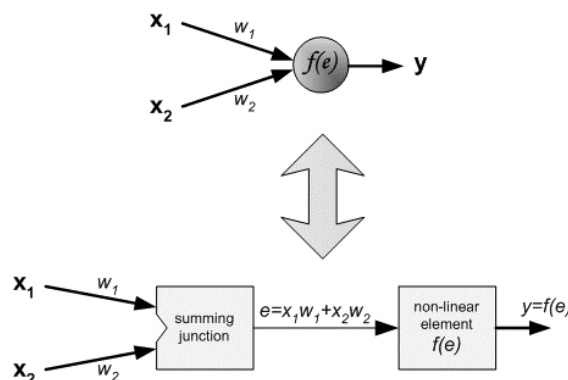


Рисунок 2.31 – Приклад математичної будови нейрону

Щоб навчити нейронну мережу ми маємо підготувати навчальні дані (приклади). У нашому випадку, тренувальні дані складаються з вхідних сигналів (x_1 та x_2) та бажаного результату z . Навчання – це послідовність ітерацій (повторень). На кожній ітерації вагові коефіцієнти нейронів підганяються з використанням нових даних із тренувальних прикладів. Зміна вагових коефіцієнтів і становить суть МЗП, описаного нижче.

Вхідний шар. Кожен крок навчання починається з впливу вхідних сигналів із тренувальних прикладів. Після цього ми можемо визначити значення вихідних сигналів для всіх нейронів у кожному шарі мережі. На рисунку 2.32 показано, як сигнал поширюється по мережі (пряме поширення). Символи $W_{(x_m)n}$ представляють вагу зв'язку між входом X_m та нейроном n у вхідному шарі. Символи $y(n)$ представляють вихідний сигнал нейрона n .

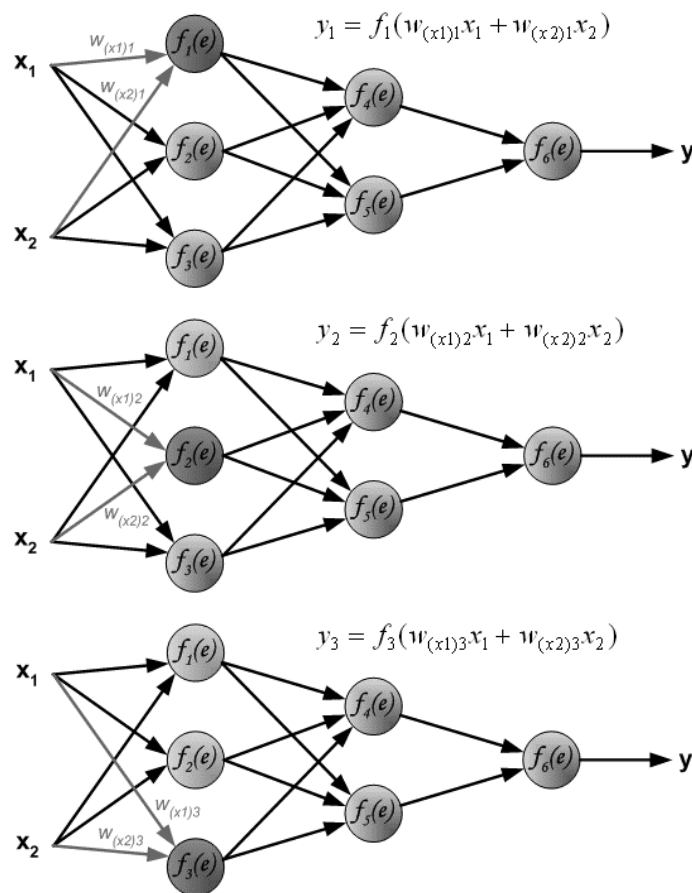


Рисунок 2.32 – Приклад прямого поширення сигналу по мережі на вхідному шарі

Розповсюдження сигналу через прихований шар. Символи w_{mn} представляють вагові множники зв'язків між виходом нейрона m та входом нейрона n у наступному шарі (рисунок 2.33).

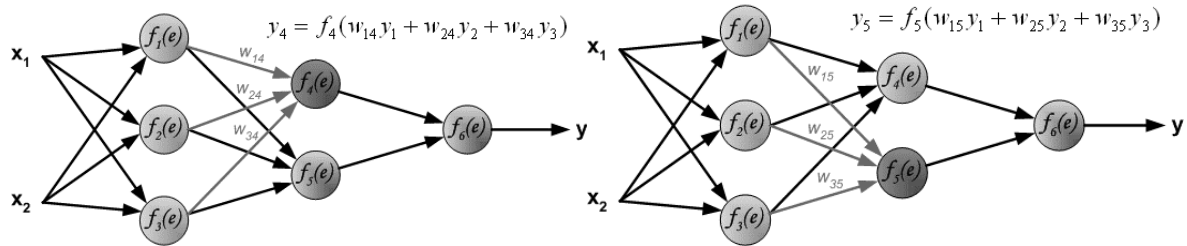


Рисунок 2.33 – Приклад прямого поширення сигналу по мережі на прихованому шарі

Поширення сигналу через вихідний шар наведено на рисунку 2.34.

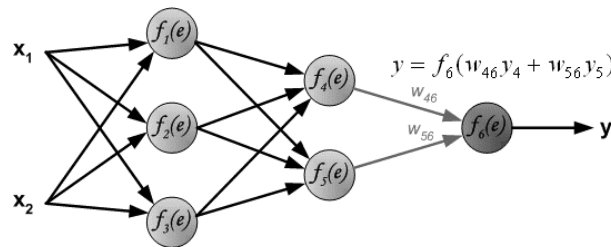


Рисунок 2.34 – Приклад прямого поширення сигналу по мережі у вихідному шарі

На наступному кроці алгоритму вихідний сигнал мережі y порівнюється з бажаним вихідним сигналом z , який зберігається в тренувальних даних. Різниця між цими двома сигналами називається помилкою вихідного шару мережі (рисунок 2.35).

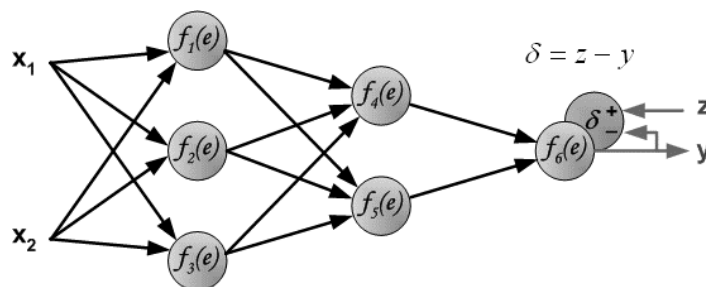


Рисунок 2.35 – Помилка вихідного шару мережі

Неможливо безпосередньо обчислити сигнал помилки для внутрішніх нейронів, тому що вихідні значення цих нейронів невідомі. Протягом багатьох років був невідомий ефективний метод навчання багатошарової мережі. Тільки в середині вісімдесятих років було розроблено алгоритм *зворотного поширення помилки*. Ідея полягає в поширенні сигналу помилки d (обчисленого під час навчання) назад на всі нейрони, вихідні сигнали яких були вхідними для останнього нейрона (рисунки 2.36).

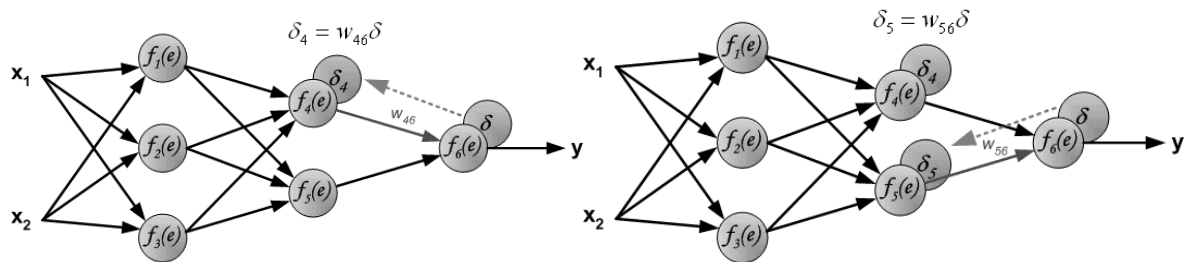
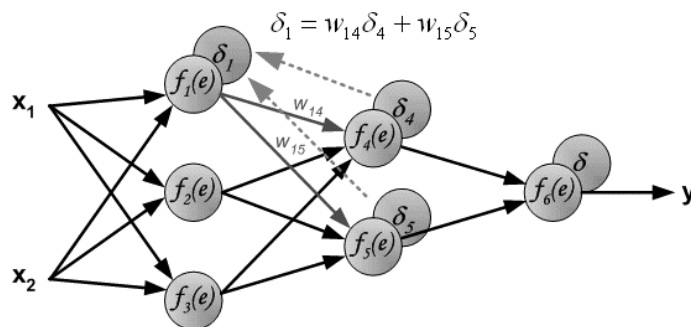


Рисунок 2.36 – Зворотнє поширення помилки з вихідного шару

Вагові коефіцієнти w_{mn} , що використовуються для зворотного поширення помилки, рівні тим самим коефіцієнтам, що використовувалися під час обчислення вихідного сигналу. Тільки змінюється напрямок потоку даних (сигнали передаються від виходу до входу). Цей процес повторюється на всіх шарах мережі. Якщо помилка прийшла від кількох нейронів – вона підсумовується (рисунки 2.37).



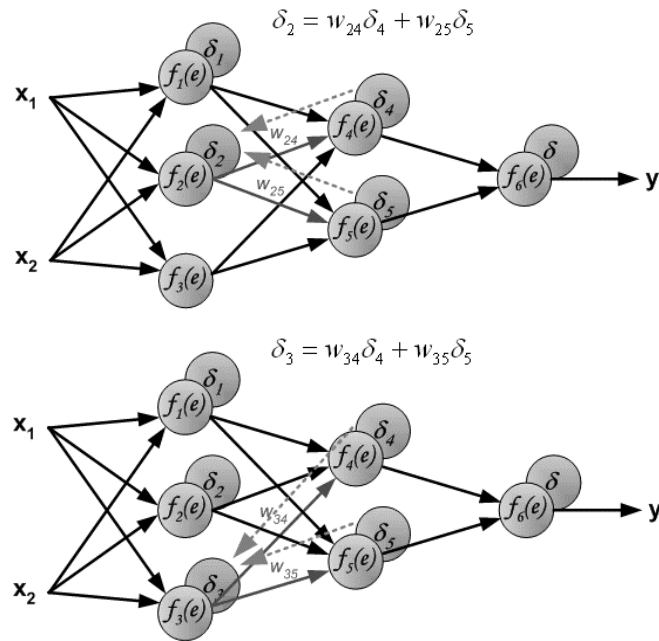


Рисунок 2.37 – Зворотнє поширення помилки для прихованого шару

Коли обчислюється величина помилки сигналу для кожного нейрона – можна скоригувати вагові коефіцієнти кожного вузла вводу (дендрита) нейрона.

У формулах нижче $df(e)/de$ є похідною від функції активації нейрона (чий вагові коефіцієнти коригуються).

Пригадаємо, що для активаційної функції типу сигмоїди похідна виражається через саму функцію

$$S'(x) = S(x) * (1 - S(x))$$

Це дозволяє істотно скоротити обчислювальну складність методу зворотного поширення помилки. Обчислення похідної необхідно, тому що для коригування вагових коефіцієнтів при навчанні ШНМ за допомогою алгоритму зворотного розповсюдження використовується метод градієнтного спуску (рисунок 2.38).

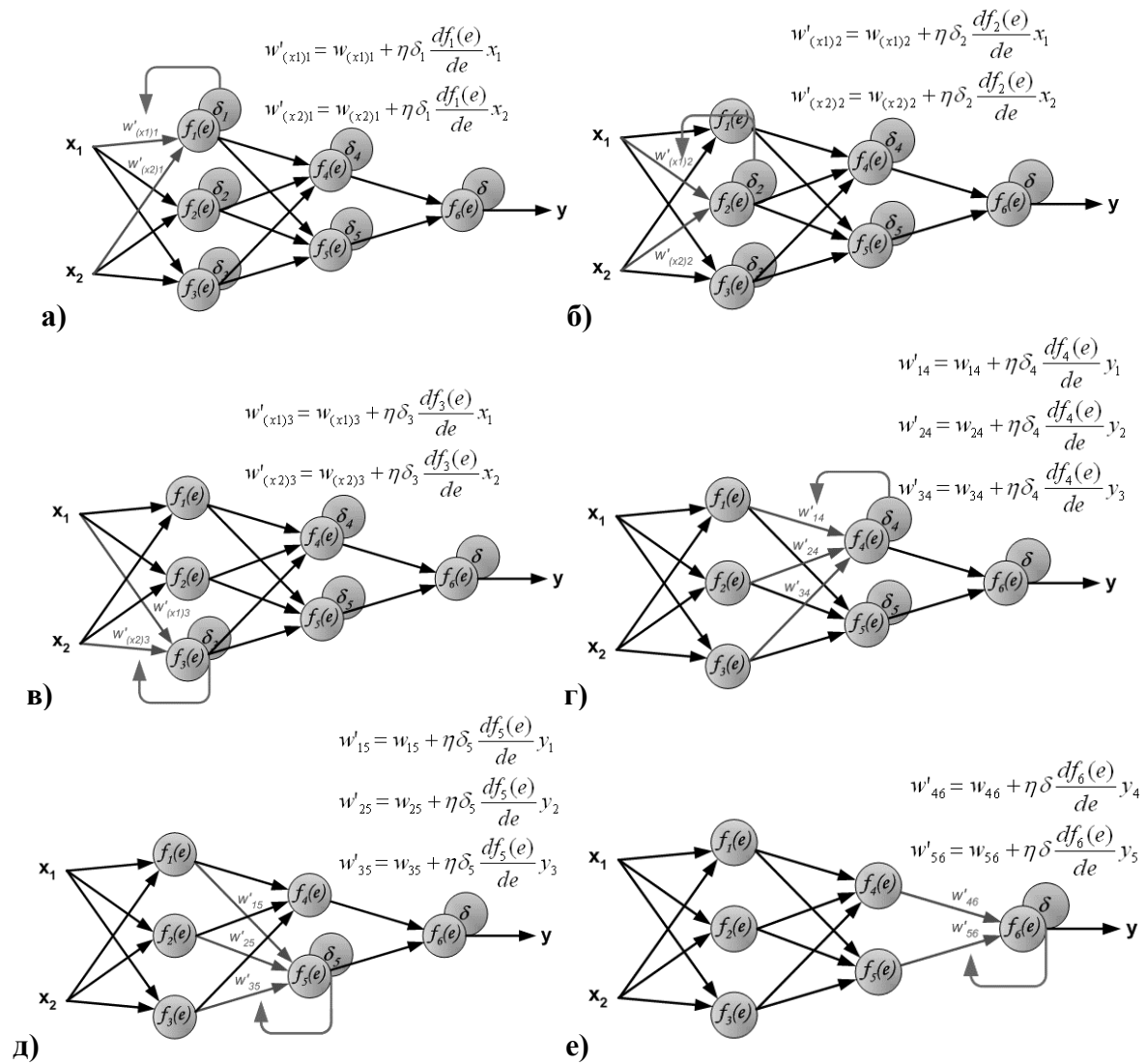


Рисунок 2.38 – Використання методу градієнтного спуску при зворотному поширенні помилки

Коефіцієнт η , як ми вже знаємо, впливає на швидкість навчання мережі. Існує кілька методів для вибору цього параметра. **Перший спосіб** – розпочати навчальний процес із великим значенням параметра η . Під час корекції вагових коефіцієнтів параметр поступово зменшують. **Другий** – складніший метод навчання, починається з малим значенням параметра η . У процесі навчання параметр збільшується, та знову зменшується на завершальній стадії навчання. Початок навчального процесу з низьким значенням параметра η дозволяє визначити знак вагових коефіцієнтів.

Отже, підсумовуючи, отримали наступний алгоритм навчання нейронної мережі, який складається з прямого та зворотного поширення помилки. Розглянемо весь алгоритм на спрощеному прикладі системи, що складається з 1 прихованого шару (рисунок 2.39)

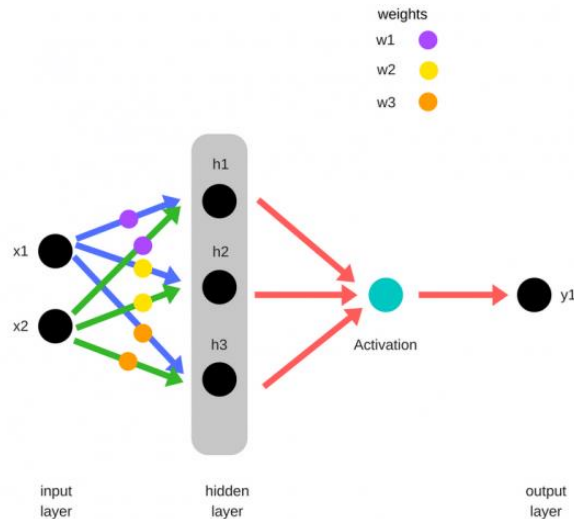


Рисунок 2.39 – Приклад одношарової нейронної мережи

Пряме поширення:

Задамо початкові ваги випадковим чином:

$$w1, w2, w3$$

Помножимо вхідні дані на ваги для формування прихованого шару:

$$h1 = (x1 * w1) + (x2 * w1)$$

$$h2 = (x1 * w2) + (x2 * w2)$$

$$h3 = (x1 * w3) + (x2 * w3)$$

Вихідні дані із прихованого шару передаються через нелінійну функцію (функцію активації), щоб отримати вихід мережі:

$$y' = f(h1, h2, h3)$$

Зворотне поширення:

- Сумарна помилка (*total_error*) обчислюється як різниця між очікуваним значенням y (з навчального набору) та отриманим значенням y' (розраховане на етапі прямого поширення помилки), що проходять через функцію втрат (*cost function*).

- Часткова похідна помилки обчислюється за кожною вагою (ці часткові диференціали відображають вклад кожної ваги у загальну помилку (*total_loss*)).
- Потім ці диференціали множаться на число, яке називається швидкість навчання або *learning rate* (η).
- Отриманий результат потім віднімається з відповідних ваг.
- В результаті вийдуть такі оновлені ваги, для спрощеного прикладу нейронної мережі (рисунок 2.40):

$$w1 = w1 - (\eta * \partial(\text{err}) / \partial(w1))$$

$$w2 = w2 - (\eta * \partial(\text{err}) / \partial(w2))$$

$$w3 = w3 - (\eta * \partial(\text{err}) / \partial(w3))$$

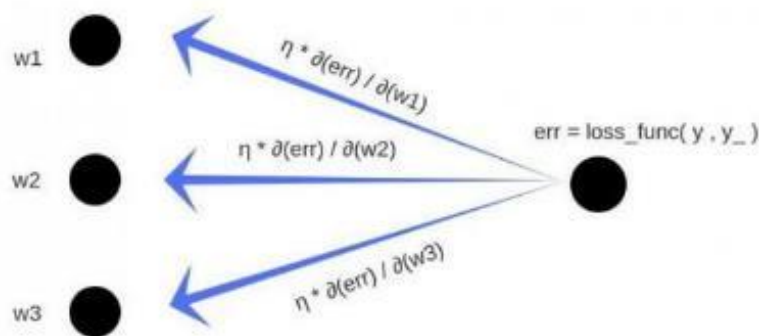


Рисунок 2.40 – Зворотне поширення помилки

На рисунку 2.41 наведено схематичне порівняння прямого та зворотного поширення помилки

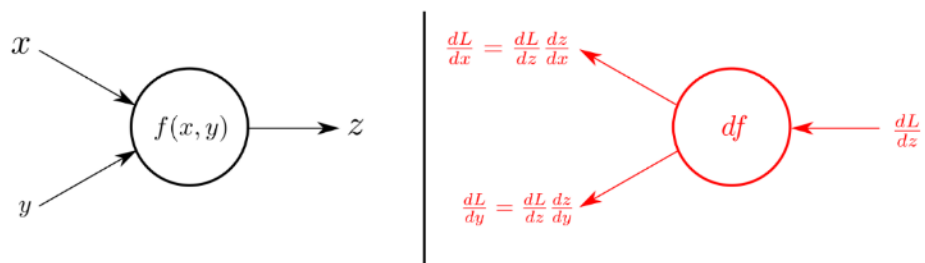


Рисунок 2.41 – Схематичне порівняння прямого (зліва) та зворотного (праворуч) поширення помилки

Методи навчання

Навчання можна проводити трьома методами: **стохастичний метод (stochastic)**, **пакетний метод (batch)** та **міні-пакетний метод (mini-batch)**. Існує дуже багато статей та досліджень на тему того, який з методів кращий і ніхто не може дати загальної відповіді. Загалом стохастичний метод використовується, мабуть, найчастіше, проте кожен метод має свої плюси та мінуси.

Стохастичний (stochastic), його ще іноді називають онлайн) метод працює за наступним принципом – знайшов Δw , відразу онови відповідну вагу. Один із найважливіших плюсів такого підходу – це значна економія часу на обчислення, точність у такому разі може сильно постраждати.

Пакетний спосіб (batch) працює інакше. Ми підсумовуємо Δw всіх ваг на поточній ітерації і лише потім оновлюємо всі ваги, використовуючи цю суму.

Міні-пакетний (mini-batch) метод є золотою серединою і намагається поєднати у собі плюси обох методів. Тут принцип такий: ми у вільному порядку розподіляємо ваги по групах і змінюємо їх ваги на суму Δw всіх ваг у тій чи іншій групі.

Що таке гіперпараметри?

Гіперпараметри – це значення, які потрібно підбирати вручну і часто методом спроб і помилок при навчанні нейронної мережі. Серед таких значень можна виділити наступні найбільш відомі:

- Величина моменту та швидкість навчання
- Кількість прихованих шарів
- Кількість нейронів у кожному шарі
- Наявність або відсутність нейронів зсуву

Підбір правильних гіперпараметрів дуже важливий і безпосередньо впливатиме на збіжність вашої нейронної мережі. Деякі гіперпараметри

підібрати просто, деякі ні. Наприклад, зрозуміти чи варто використовувати нейрони зсуву чи ні у архітектурі вашої мережі досить просто. Кількість прихованих шарів і нейронів у них можна обчислити перебором, ґрунтуючись на одному простому правилі – чим більше нейронів, тим точніше результат і тим експоненційно більший час, який ви витратите на її навчання. Однак варто пам'ятати, що не варто робити ШНМ з 1000 нейронів для вирішення простих завдань. А ось з вибором моменту та швидкості навчання все трішки складніше. Ці гіперпараметри будуть змінюватись, залежно від поставленого завдання та архітектури мережі. Наприклад, для побудови моделі XOR елементу швидкість навчання може бути в межах 0.3 - 0.7, але в нейронній мережі, яка аналізує і передбачає ціну акцій, швидкість навчання вище 0.00001 призводить до поганої збіжності ШНМ. Не варто зараз загострювати свою увагу на гіперпараметрах і намагатися досконало зрозуміти, як їх вибирати. Це прийде з досвідом, а поки що раджу просто експериментувати та шукати приклади вирішення того чи іншого завдання методом побудови нейронної мережі.

Що таке збіжність?

Збіжність моделі говорить про те, чи правильна архітектура ШНМ і чи правильно були підібрані гіперпараметри відповідно до поставленого завдання. Припустимо наша програма виводить помилку НМ на кожній ітерації. Якщо з кожною ітерацією помилка буде зменшуватися, то ми на вірному шляху і наша мережа сходиться. Якщо ж помилка стрибатиме вгору – вниз або застигне на певному рівні, то НМ не сходиться. У 99% випадків це вирішується зміною гіперпараметрів. 1%, що залишився, означатиме, що у вас помилка в архітектурі НМ. Також буває, що на збіжність впливає перенавчання НМ (рисунок 2.42).

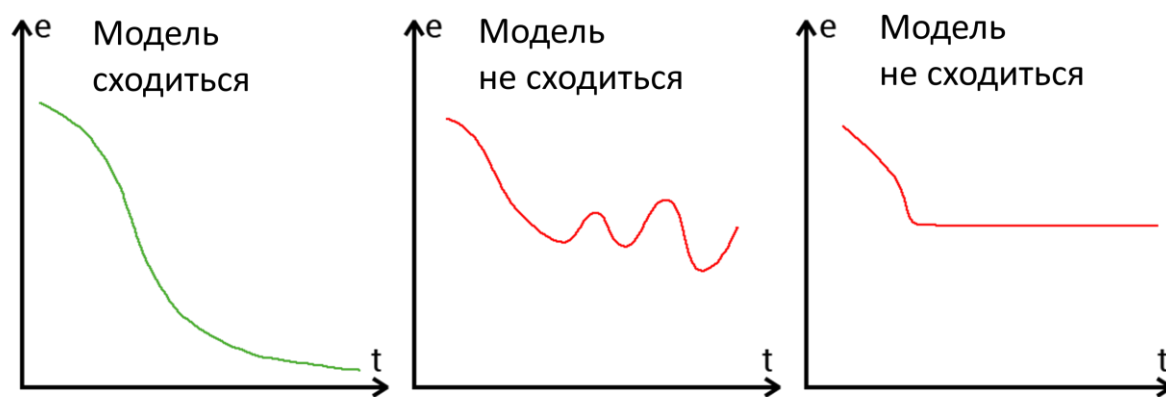


Рисунок 2.42 – Сходимость моделі

Різноманітність нейронних мереж.

Нові види архітектури нейронних мереж виникають постійно, і в них можна заплутатися, хоча багато з них насправді дуже схожі одна на одну. На рисунку 2.14 наведена повна ієрархія архітектур та позначення, що будуть використані далі [4-18].

Нейронні мережі прямого розповсюдження.

Нейронні мережі прямого розповсюдження (feed forward neural networks, FF чи FFNN) та перцептрони (perceptrons, P) дуже прямолінійні, вони передають інформацію від входу до виходу (рисунок 2.43). Нейронні мережі часто описуються у вигляді торта, де кожен шар складається з вхідних, прихованих або вихідних клітин. Клітини одного шару пов'язані між собою, а сусідні шари зазвичай повністю пов'язані. Найпростіша нейронна мережа має дві вхідні клітини і одну вихідну, і може використовуватися як модель логічних вентилів. FFNN зазвичай навчається за методом зворотного поширення помилки, в якому мережа отримує безліч вхідних та вихідних даних. Ця помилка є різницею між вхідними та вихідними даними. Якщо мережа має достатню кількість прихованих нейронів, вона теоретично здатна змодельовати взаємодію між вхідним і вихідними даними. На практиці такі мережі використовуються рідко, але часто їх комбінують з іншими типами для отримання нових архітектур.



Рисунок 2.43 – Схематичне представлення мережі прямого розповсюдження

Мережі радіально-базових функцій (Radial basis function, RBF) – це FFNN, яка використовує радіальні базисні функції як функції активації. Більше вона нічим не вирізняється.

Нейронна мережа Хопфілда (Hopfield network, HN) – це повнозв'язна нейронна мережа із симетричною матрицею зв'язків (рисунок 2.44). Під час отримання вхідних даних кожен вузол є входом, у процесі навчання стає прихованим, а потім стає виходом. Мережа навчається так: значення нейронів встановлюються відповідно до бажаного шаблону, після чого обчислюються ваги, які надалі не змінюються. Після того, як мережа навчилася на одному або декількох шаблонах, вона завжди зводиться до одного з них (але не завжди до бажаного). Вона стабілізується залежно від загальної «енергії» та «температури» мережі. У кожного нейрона є свій поріг активації, що залежить від температури, при проходженні якого нейрон набуває одного з двох значень (зазвичай -1 або 1, іноді 0 або 1). Така мережа часто називається мережею з асоціативною пам'яттю; як людина, що бачить половину таблиці, може уявити другу половину таблиці, і ця мережа, отримуючи таблицю, наполовину з шумом, відновлює її до повної.



Рисунок 2.44 – Схематичне представлення мережі Хопфілда

Ланцюги Маркова (Markov chains, MC або discrete time Markov Chains, DTMC) – це попередники машин Больцмана та мереж Хопфілда (рисунок 2.45). Їхній сенс можна пояснити так: які мої шанси потрапити в один із наступних вузлів, якщо я перебуваю в цьому? Кожен наступний стан залежить лише від попереднього. Хоча насправді ланцюги Маркова є нейронною мережею, вони дуже схожі. Також ланцюги Маркова не обов'язково повні.

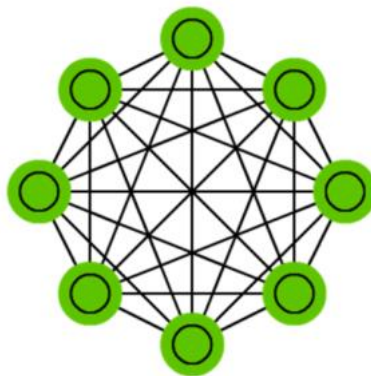


Рисунок 2.45 – Схематичне представлення Ланцюгів Маркова

Машина Больцмана (Boltzmann machine, BM) дуже схожа на мережу Хопфілда, але в ній деякі нейрони позначені як вхідні, а деякі як приховані (рисунок 2.46). Вхідні нейрони надалі стають вихідними. Машина Больцмана – це стохастична мережа. Навчання відбувається за методом зворотного поширення помилки або алгоритмом порівняльної розбіжності. Загалом процес навчання дуже схожий на навчання у мережі Хопфілда.



Рисунок 2.46 – Схематичне представлення Мащини Больцмана

Обмежена машина Больцмана (restricted Boltzmann machine, RBM) напрочуд схожа на машину Больцмана і, отже, на мережу Хопфілда (рисунок 2.47). Єдиною різницею є її обмеженість. У ній нейрони одного типу пов'язані між собою. Обмежену машину Больцмана можна навчати як FFNN, але з одним нюансом: замість прямої передачі даних та зворотного розповсюдження помилки потрібно передавати дані спершу у прямому напрямку, потім у зворотному. Після цього проходить навчання за методом прямого та зворотного поширення помилки.

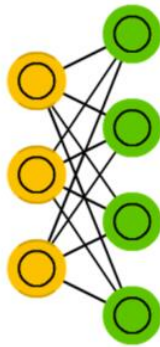


Рисунок 2.47 – Схематичне представлення Обмеженої машини Больцмана

Автокодувальник (autoencoder, AE) чимось схожий на FFNN, оскільки це скоріш інший спосіб використання FFNN, ніж фундаментально інша архітектура. Основною ідеєю є автоматичне кодування (у сенсі стиснення, не шифрування) інформації. Сама мережа формою нагадує пісочний годинник, у ній приховані шари менше вхідного і вихідного, причому вона симетрична. Мережу можна навчити методом зворотного поширення помилки, подаючи вхідні дані та задаючи помилку рівною різниці між входом та виходом.

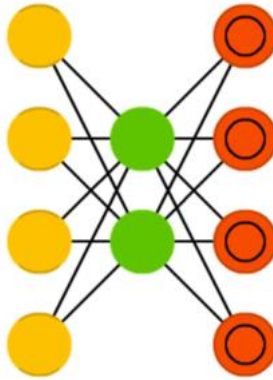


Рисунок 2.48 – Схематичне представлення Автокодувальника

Розріджений автокодувальник (sparse autoencoder, SAE) – у певному сенсі протилежність звичайного (рисунок 2.49). Замість того, щоб навчати мережу відображати інформацію в меншому обсязі вузлів, ми збільшуємо їх кількість. Замість того, щоб звужуватись до центру, мережа там збільшується. Мережі такого типу корисні для роботи з великою кількістю дрібних властивостей набору даних. Якщо навчати мережу як звичайний автокодувальник, нічого корисного не вийде. Тому, крім вхідних даних, подається ще й спеціальний фільтр розрідженості, який пропускає лише певні помилки.

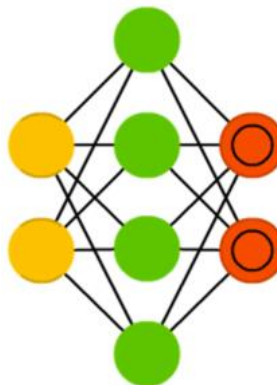


Рисунок 2.49 – Схематичне представлення Розрідженого Автокодувальника

Варіаційні автокодувальники (variational autoencoder, VAE) мають схожу з АЕ архітектурою, але навчають їх по іншому: методом наближення ймовірнісного розподілу вхідних зразків (рисунок 2.50). У цьому вони

беруть початок від машин Больцмана. Тим не менш, вони спираються на байєсовську математику, коли йдеться про ймовірні висновки та незалежності, які інтуїтивно зрозумілі, але складні в реалізації. Якщо узагальнити, то можна сказати, що ця мережа бере до уваги вплив нейронів. Якщо щось одне відбувається в одному місці, а щось інше — в іншому, ці події не обов'язково пов'язані, і це має враховуватися.

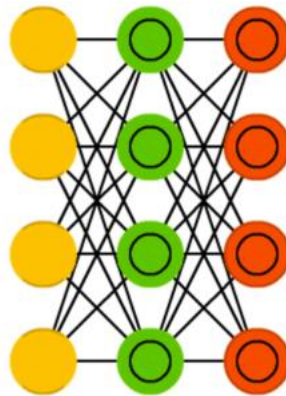


Рисунок 2.50 – Схематичне представлення Розрідженого Автокодувальника

Шумопригнічуючі автокодувальники (denoising autoencoder, DAE) – це АЕ, в які вхідні дані подаються в зашумленому стані (рисунок 2.51). Помилки ми обчислюємо так само, і вихідні дані порівнюються із зашумленими. Завдяки цьому мережа вчиться звертати увагу на більш широкі властивості, оскільки маленькі можуть змінюватися разом із шумом.

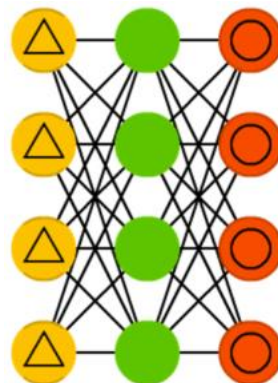


Рисунок 2.51 – Схематичне представлення Шумопригнічуючого автокодувальника

Мережа типу "deep belief" (deep belief networks, DBN) – це назва, яка отримала тип архітектури, в якій мережа складається з декількох з'єднаних RBM або VAE (рисунок 2.52). Такі мережі навчаються поблочно, причому кожному блоку потрібно лише вміти закодувати попередній. Така техніка називається «жадібним навчанням», яка полягає у виборі локальних оптимальних рішень, що не гарантують оптимальний кінцевий результат. Також мережу можна навчити (методом зворотного поширення помилки) відображати дані у вигляді імовірнісної моделі. Якщо використовувати навчання без вчителя, стабілізовану модель можна використовувати для створення нових даних.

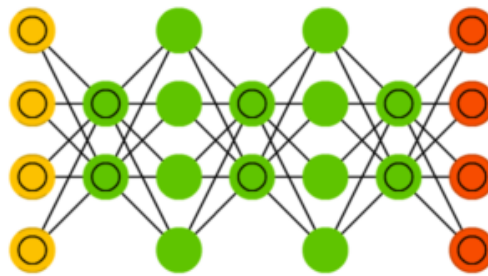


Рисунок 2.52 – Схематичне представлення deep belief networks

Згорткові нейронні мережі (convolutional neural networks, CNN) та **глибинні згорткові нейронні мережі** (deep convolutional neural networks, DCNN) сильно відрізняються від інших видів мереж (рисунок 2.53). Зазвичай вони використовуються для обробки зображень, рідше для аудіо. Типовим способом застосування CNN є класифікація зображень: якщо на зображенні є кішка, мережа видає кішка, якщо є собака – собака. Такі мережі зазвичай використовують «сканер», який не парсить всі дані за один раз. Наприклад, якщо є зображення 200×200 , то не треба відразу обробляти всі 40 тисяч пікселів. Замість цього мережа рахує квадрат розміру 20×20 (зазвичай з лівого верхнього кута), потім робить зсув на 1 піксель і рахує новий квадрат, і т.д. Ці вхідні дані передаються через згорткові шари, в яких не всі вузли з'єднані між собою. Ці шари мають властивість стискатися з

глибиною, причому часто використовуються ступені двійки: 32, 16, 8, 4, 2, 1. На практиці до кінця CNN прикріплюють FFNN для подальшої обробки даних. Такі мережі називаються глибокими (DCNN).

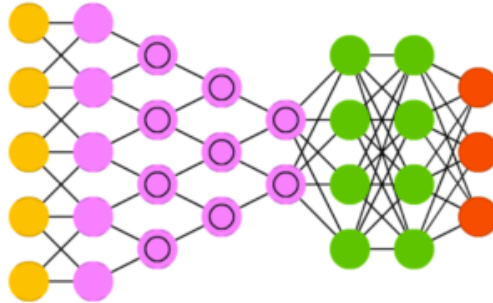


Рисунок 2.53 – Схематичне представлення згорткових нейронних мереж

Нейронні мережі, що розгортають (deconvolutional networks, DN), також звані зворотними графічними мережами, є зворотними до згорткових нейронних мереж (рисунок 2.54). Уявіть, що ви передаєте мережі слово "кішка", а вона генерує картинки з кішками, схожі на реальні зображення котів. DNN також можна поєднувати з FFNN. Варто зауважити, що у більшості випадків мережі передається не рядок, а якийсь бінарний вектор: наприклад, $\langle 0, 1 \rangle$ - це кішка, $\langle 1, 0 \rangle$ - собака, а $\langle 1, 1 \rangle$ - і кішка, і собака.

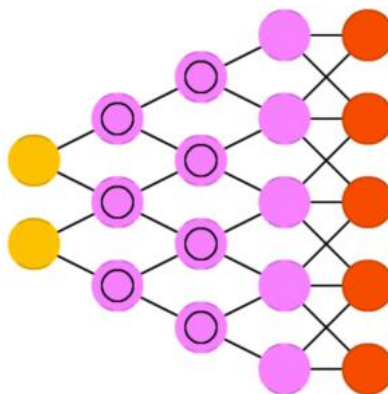


Рисунок 2.54 – Схематичне представлення нейронних мереж, що згортають

Глибинні згорткові зворотні графічні мережі (deep convolutional inverse graphics networks, DCIGN) названі злегка некоректно, оскільки вони

по суті є варіаційними автокодувальниками, що кодує та декодує частини яких представлені згортковою та розгортаючою нейромережею, відповідно (рисунок 2.55). Мережі такого типу моделюють властивості у вигляді ймовірностей, тому їх можна навчити створювати картинку із собакою та кішкою, навіть якщо мережа бачила лише картинки, на яких була лише одна з тварин. Можливе видалення одного з двох об'єктів. Також були створені мережі, які могли змінювати джерело освітлення та обертати об'єкт. Мережі такого типу зазвичай навчають шляхом зворотного поширення помилки.

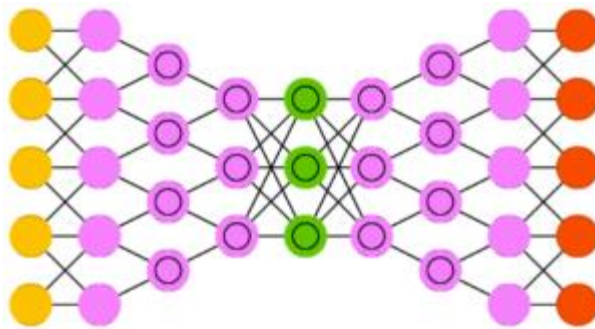


Рисунок 2.55 – Схематичне представлення глибинних згорткових зворотних графічних мереж

Генеративні змагальні мережі (generative adversarial networks, GAN) – це мережі іншого виду, вони схожі на близнюків (рисунок 2.56). Такі мережі складаються з будь-яких двох мереж (зазвичай з FF та CNN), одна з яких контент генерує, а інша оцінює. Мережа-дискримінатор отримує навчальні чи створені генератором дані. Ступінь вгадування дискримінатором джерела даних надалі бере участь у формуванні помилки. Таким чином, виникає змагання між генератором і дискримінатором, де перший вчиться обманювати першого, а другий розкривати обман (схоже на ситуацію «банкір-фальшивомонетник»). Навчати такі мережі дуже важко, оскільки потрібно не лише навчити кожну з них, а й настроїти баланс.

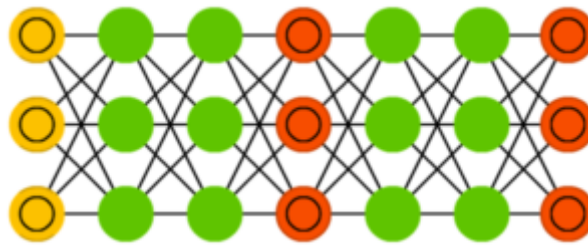


Рисунок 2.56 – Схематичне представлення генеративно змагальних мереж

Рекурентні нейронні мережі (recurrent neural networks, RNN) – це мережі типу FFNN, але з особливістю: нейрони отримують інформацію не тільки від попереднього шару, а й від себе від попереднього проходу (рисунок 2.57). Це означає, що порядок, в якому ви подаєте дані та навчаєте мережу, стає важливим. Великою складністю мереж RNN є проблема зникаючого (або вибухового) градієнта, яка полягає у швидкій втраті інформації з часом. Звичайно, це впливає лише на ваги, а не стан нейронів, але саме в них накопичується інформація. Зазвичай, мережі такого типу використовуються для автоматичного доповнення інформації.

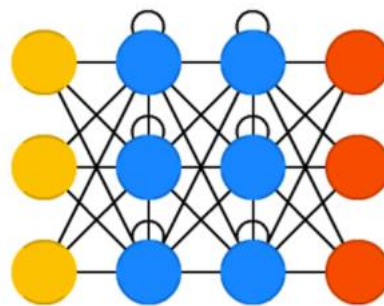


Рисунок 2.57 – Схематичне представлення рекурентних мереж

Мережі з довгою короткостроковою пам'яттю (long short term memory, LSTM) намагаються вирішити вищезгадану проблему втрати інформації, використовуючи фільтри та явно задану комірку пам'яті (рисунок 2.58). У кожного нейрона є комірка пам'яті і три фільтри: вхідний, вихідний і забуваючий. Метою цих фільтрів є захист інформації. Вхідний фільтр визначає, скільки інформації з попереднього шару зберігатиметься в комірці. Вихідний фільтр визначає, скільки інформації отримають такі

шари. Ну а фільтр, що забуває, яким би дивним не здавався, також виконує корисну функцію: наприклад, якщо мережа вивчає книгу і переходить на новий розділ, якісь символи зі старої можна забути. Такі мережі здатні навчитися створювати складні структури, наприклад, писати як Шекспір чи складати просту музику, але й ресурсів вони споживають чимало.

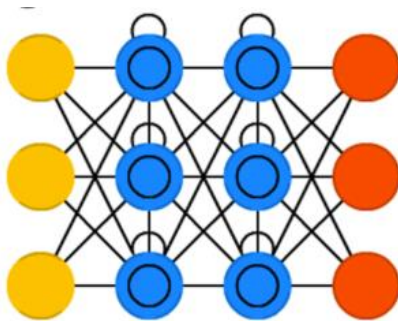


Рисунок 2.58 – Схематичне представлення мережі з довгою короткостроковою пам'яттю

Керовані рекурентні нейрони (gated recurrent units, GRU) – це невелика варіація попередньої мережі (рисунок 2.59). У них на один фільтр менше, і зв'язки реалізовані трохи інакше. Фільтр оновлення визначає, скільки інформації залишиться від минулого стану та скільки буде взято з попереднього шару. Фільтр скидання працює приблизно як фільтр, що забуває.

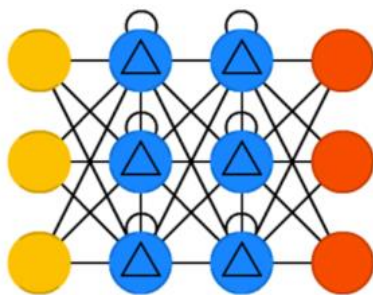


Рисунок 2.59 – Схематичне представлення GRU мережі

Нейронні машини Тьюринга (neural Turing machines, NTM) можна розглядати як абстрактну модель LSTM і спробу показати, що насправді відбувається усередині нейронної мережі (рисунок 2.60). Комірка пам'яті не

поміщена в нейрон, а розміщена окремо з метою об'єднати ефективність звичайного сховища даних та потужність нейронної мережі. Власне, тому такі мережі і називаються машинами Тьюрінга — через здатність читати та записувати дані та змінювати стан залежно від прочитаного вони є тьюрінг-повними.

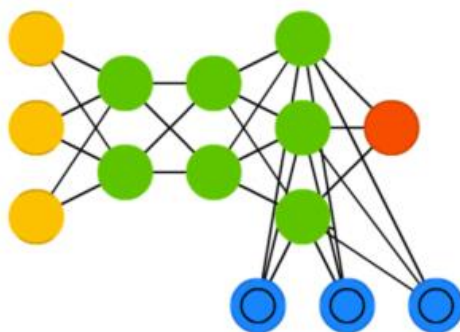


Рисунок 2.60 – Схематичне представлення нейронних машин Тьюрінга

Двонаправлені RNN, LSTM і GRU (bidirectional recurrent neural networks, bidirectional long / short term memory networks і bidirectional gated recurrent units, BiRNN, BiLSTM і BiGRU) – вони нічим не відрізняються від своїх односпрямованих варіантів. Різниця полягає в тому, що ці мережі використовують не лише дані з «минулого», а й із «майбутнього». Наприклад, звичайну мережу типу LSTM навчають вгадувати слово "риба", подаючи літери по одній, а двонаправлену – подаючи ще й наступну літеру з послідовності. Такі мережі здатні, наприклад, не лише розширювати зображення по краях, а й заповнювати дірки всередині.

Глибинні залишкові мережі (deep residual networks, DRN) – це дуже глибокі мережі типу FFNN з додатковими зв'язками між відокремленими один від одного шарами (рисунок 2.61). Такі мережі можна навчати на шаблонах глибиною аж до 150 шарів – набагато більше, ніж можна було б очікувати. Однак було показано, що ці мережі мало чим відрізняються від рекурентних, і їх часто порівнюють з мережами LSTM.

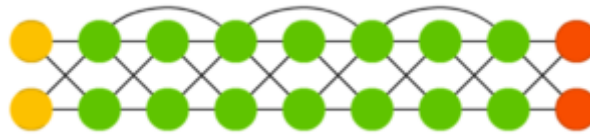


Рисунок 2.61 – Схематичне представлення глибоких залишкових нейронних мереж

Нейронна ехо-мережа (echo state networks, ESN) – це ще один різновид рекурентних мереж (рисунок 2.62). Її особливістю є відсутність сформованих шарів, тобто зв'язки між нейронами випадкові. Відповідно, метод зворотного поширення помилки не спрацьовує. Натомість потрібно подавати вхідні дані, передавати їх по мережі та оновлювати нейрони, спостерігаючи за вихідними даними.

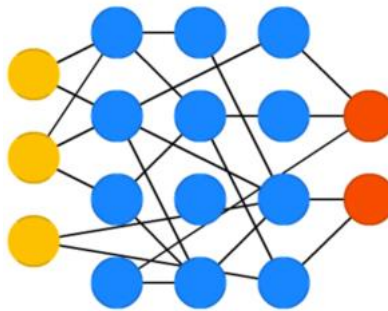


Рисунок 2.62 – Схематичне представлення нейронних ехо-мереж

Метод екстремального навчання (extreme learning machines, ELM) – це, по суті, мережа типу FFNN, але із випадковими зв'язками (рисунок 2.63). Вони дуже схожі на мережі LSM та ESN, але використовуються як FFNN. Так відбувається не тільки тому, що вони не рекурентні, а й тому, що їх можна навчати просто шляхом зворотного розповсюдження помилки.

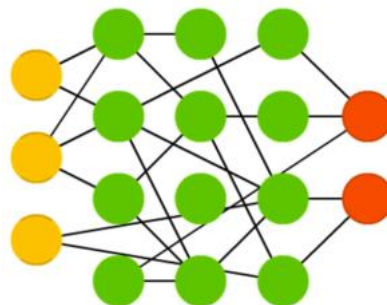


Рисунок 2.63 – Схематичне представлення мережі екстремального навчання

Метод нестійких станів (liquid state machines, LSM) схожий на ехо-мережу, але є суттєва відмінність: сигмоїдна активація замінена пороговою функцією, а кожен нейрон є накопичувальним осередком пам'яті (рисунок 2.64). Таким чином, при оновленні нейрона його значення не стає рівним сумі сусідів, а додається саме до себе, і при досягненні порога повідомляється іншим нейронам.

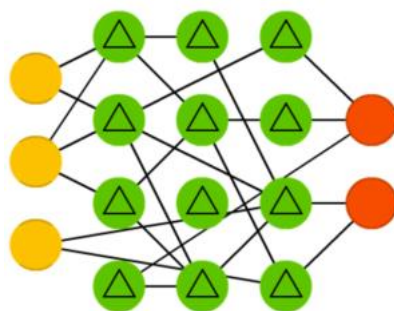


Рисунок 2.64 – Схематичне представлення методу нестійких станів

Метод опорних векторів (support vector machines, SVM) знаходить оптимальні рішення задачі оптимізації. Класична версія здатна категоризувати дані, що лінійно розділяються: наприклад, розрізнити зображення з котом Томом і з котом Гарфілдом. В процесі навчання мережа як би розміщує всі дані на 2D-графіці і намагається розділити дані прямою лінією так, щоб з кожного боку були дані лише одного класу і щоб відстань від даних до лінії була максимальною. Використовуючи трюк із ядром, можна класифікувати дані розмірності n . Що характерно, цей метод не завжди сприймається як нейронна мережа.

І нарешті, **нейронні мережі Кохонена** (Kohonen networks, KN), також відомі як карти, що самоорганізуються (self organising (feature) maps, SOM, SOFM), завершують наш список (рисунок 2.65). Ці мережі використовують змагання для класифікації даних без вчителя. Мережі подаються вхідні дані,

після чого мережа визначає які з нейронів максимально збігаються з ними. Після цього ці нейрони змінюються для ще більшої точності збігу, в процесі рухаючи за собою сусідів. Іноді карти Кохонена також вважаються нейронними мережами.

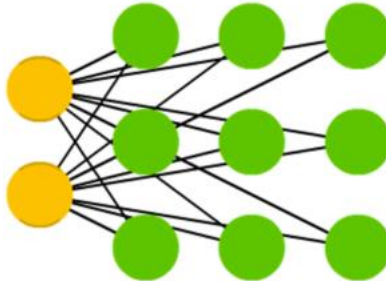


Рисунок 2.65 – Схематичне представлення нейронних мереж Кохонена

Контрольні запитання

1. Що таке нейрон?
2. З цього складається типовий нейрон?
3. Яка функція основних структурних елементів математичної моделі нейрона?
4. Що таке перцептрон?
5. Які бувають функції активації

Перелік посилань

1. <http://www.asimovinstitute.org/neural-network-zoo>

ТЕМА №3: ЗАДАЧА КЛАСИФІКАЦІЇ. ФУНКЦІЇ ВТРАТ ЗАДАЧ КЛАСИФІКАЦІЇ.

Завдання класифікації.

Класифікація – один із розділів машинного навчання, присвячений вирішенню наступного завдання. Є безліч об'єктів (ситуацій), поділених певним чином класи. Задано кінцева безліч об'єктів, для яких відомо, до яких класів вони відносяться. Ця множина називається *навчальною вибіркою*. Класова приналежність інших об'єктів невідома. Потрібно побудувати алгоритм, здатний класифікувати довільний об'єкт із вихідної множини [14-16].

Класифікувати об'єкт – означає вказати номер (або найменування класу), до якого належить даний об'єкт.

Класифікація об'єкта – номер або найменування класу, що видається алгоритмом класифікації внаслідок його застосування до конкретного об'єкта.

Іншими словами, *класифікаційне прогнозне моделювання* – це завдання апроксимації функції відображення (f) входних змінних (x) у дискретні вихідні змінні (y). Вихідні змінні часто називають *мітками чи категоріями*. Функція відображення передбачає клас або категорію для цього спостереження.

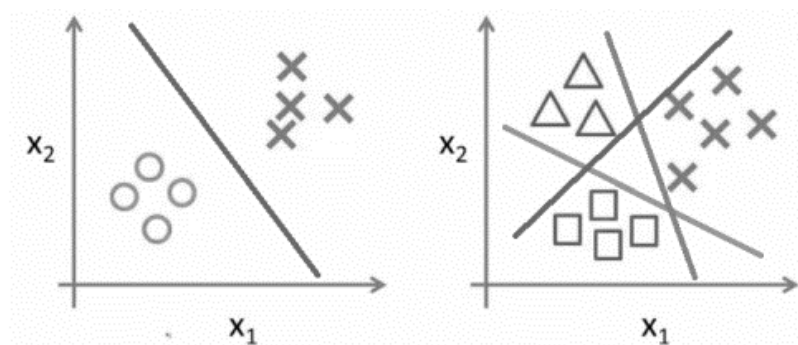


Рисунок 3.1 – Приклад бінарної (зліва) та багатокласової класифікації (праворуч)

Приклад. Допустимо у нас є зображення, і ми хочемо дізнатися чи є кіт на зображенні чи ні (рисунок 3.2). Помістимо зображення в мережу, його можна віднести до одного з двох типів: кіт або ні. *Це називається двокласовою класифікацією або двійковою класифікацією (бінарною).* З іншого боку, коли проблема має більше двох класів, у цьому випадку ми хочемо класифікувати на кішку, собаку або будь-яку іншу групу, це називається *проблемою класифікації декількох класів (багатокласовою).* Ймовірності можна інтерпретувати як ймовірність чи достовірність приналежності цього прикладу до кожного класу. Прогнозована ймовірність може бути перетворена на значення класу, вибравши позначку класу, яка має найбільшу ймовірність. У машинному навчанні завдання класифікації належить до розділу *навчання з учителем.*

Нехай у наборі даних, що використовується для виявлення кішки – сотні зображень кішки та собаки змішані. З кожним зображенням кішки ми позначаємо його як 1 та 0, якщо ні. Наша місія проста: помістити зображення в мережу, вона поверне результат у вигляді плаваючого числа, що дозволяє передбачити, до якого класу зображення належить. Якщо результат дорівнює 1, не можна заперечувати, що це кішка, і, навпаки. Однак нейронна мережа не працює таким чином, і ми отримуємо результат який є дійсним числом, наприклад 0.1, 0.5 або 0.8. І звідси ми визначаємо, 0.5 чи 0.8 – кішка чи ні.

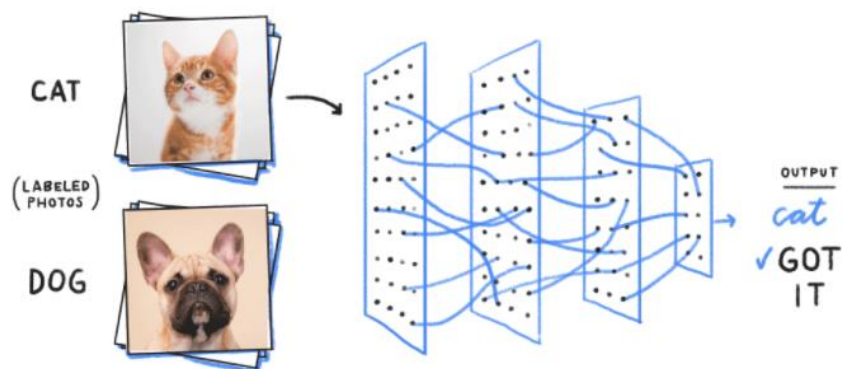


Рисунок 3.2 – Приклад задачі класифікації «кіт – собака»

Очевидно, ми можемо зробити висновок, що 0.8 ближче до 1, тому, якщо зображення повертає 0.8, ймовірність того, що це кішка, вище ніж 0.5. Але нейронна мережа не така проста, гірка правда в тому, що будуть деякі випадки, які повертають 0.5, навіть 0.1, але це все ще кіт. Навпаки, 0.8, навіть 0.85 чи 0.999 не завжди буде стовідсотково означати, що це кішка. Ми вже знаємо про зворотне розповсюдження помилки для налаштування параметрів ШНМ. Але перед цим процедура оцінки вимагає рішення, щоб «адаптувати чи виправити» отриманий результат із тим значенням, яким він має бути. І тоді народжується *функція втрат*, види яких ми вже розглядали для задачі регресії у попередньому розділі.

Типологія завдань класифікації [5-8, 12-14]

Типи вхідних даних.

- Ознаковий опис – це найпоширеніший випадок. Кожен об'єкт описується набором своїх характеристик, які називають ознаками. Ознаки можуть бути числовими чи нечисловими.
- Матриця відстаней між об'єктами. Кожен об'єкт описується відстанями до решти об'єктів навчальної вибірки. З цим типом вхідних даних працюють деякі методи, зокрема, метод найближчих сусідів, метод парзенівського вікна, метод потенційних функцій.
- Тимчасовий ряд або сигнал є послідовністю вимірювань у часі. Кожен вимір може бути числом, вектором, а у загальному випадку – ознаковим описом досліджуваного об'єкта у даний час.
- Зображення або відеоряд.
- Зустрічаються і складніші випадки, коли вхідні дані представляються у вигляді графів, текстів, результатів запитів до бази даних, тощо. Зазвичай, вони приводяться до першого чи другого випадку шляхом попередньої обробки даних і вилучення ознак.

Класифікацію сигналів та зображень називають також *розпізнаванням образів*.

Типи класів

- Двокласова класифікація. Найбільш простий у технічному відношенні випадок, який є основою для вирішення складніших завдань.

Варто зазначити, що алгоритми, розроблені для бінарної класифікації, можуть бути адаптовані для використання в мультикласових задачах.

Це включає використання стратегії підбору декількох моделей бінарної класифікації для кожного класу в порівнянні з усіма іншими класами (*one-vs-rest – встановіть одну модель бінарної класифікації для кожного класу порівняно з іншими класами*) або однієї моделі для кожної пари класів (*one-vs-one – встановіть одну модель бінарної класифікації для кожної пари класів*).

Алгоритми двійкової класифікації, які можуть використовувати ці стратегії для мультикласової класифікації, включають: логістичну регресію, метод опорних векторів.

- Багатокласова класифікація. Коли число класів може досягати багатьох тисяч (наприклад, при розпізнаванні ієрогліфів або зливої мови), завдання класифікації стає значно важчим.
- Непересічні класи.
- Пересічні класи. Об'єкт може відноситися одночасно до кількох класів.
- Нечіткі класи. Потрібно визначати ступінь схожості об'єкта кожному з класів, зазвичай це дійсне число від 0 до 1.

У завданнях машинного навчання ознаки (features) поділяються на такі типи:

- бінарна ознака – 0 або 1;
- номінальна ознака – кінцева множина;
- порядкова ознака – кінцева впорядкована множина;

- кількісна ознака – безліч дійсних чисел.

Часто зустрічаються прикладні завдання з різнотипними ознаками, для їх вирішення підходять далеко не всі методи.

Приклади прикладних завдань [4-25]

- Завдання медичної діагностики. У ролі об'єктів виступають пацієнти. Ознаки характеризують результати обстежень, симптоми захворювання та методи лікування, що застосовувалися. Приклади бінарних ознак: стать, наявність головного болю, слабкості. Порядкова ознака – тяжкість стану (задовільний, середньої тяжкості, важкий, вкрай важкий). Кількісні ознаки – вік, пульс, артеріальний тиск, вміст гемоглобіну у крові, доза препарату. Ознаковий опис пацієнта є, по суті, формалізованою історією хвороби. Накопичивши достатню кількість прецедентів в електронному вигляді, можна вирішувати різні завдання:

- класифікувати вид захворювання (диференціальна діагностика);
- визначати найбільш доцільний спосіб лікування;
- передбачати тривалість та результат захворювання;
- оцінювати ризик ускладнень;
- знаходити синдроми – найбільш характерні для даного захворювання сукупності симптомів.

Цінність таких систем у тому, що вони здатні миттєво аналізувати і узагальнювати величезну кількість прецедентів – можливість, недоступна фахівцеві-лікарю.

- Передбачення родовищ корисних копалин
- Оцінювання кредитоспроможності позичальників
- Передбачення відтоку клієнтів
- Оптичне розпізнавання символів
- Розпізнавання мови

- Виявлення спаму
- Класифікація документів

Функції втрат задач класифікації

Алгоритм класифікації може передбачати безперервне значення, але безперервне значення має форму ймовірності для позначки класу. Алгоритм регресії може прогнозувати дискретне значення, але дискретне значення у вигляді цілого числа [13-15].

Деякі алгоритми можуть використовуватися для обох методів з невеликими модифікаціями, наприклад, дерева рішень та штучні нейронні мережі.

Важливо відзначити, що способи оцінки прогнозів класифікації та регресії розрізняються і не перетинаються, наприклад:

- Прогнози класифікації можуть бути оцінені з використанням точності, тоді як передбачення регресії – ні.
- Прогнози регресії можуть бути оцінені з використанням середньоквадратичної помилки, тоді як передбачення класифікації не можуть.

Розглянемо детальніше види функції втрат для задачі класифікації (загальний перелік функцій втрат задач регресії та класифікації наведено на рисунку 2.2).

Функція втрат перехресна (крес-) ентропія, Cross-Entropy Loss (or Log Loss).

Вона вимірює продуктивність моделі класифікації, виходом якої є значення ймовірності від 0 до 1. Втрати перехресної ентропії зростають, коли прогнозована ймовірність відхиляється від фактичної мітки. Таким чином, прогнозувати ймовірність 0.012, коли фактична мітка спостереження дорівнює 1, було б погано і призвело б до високого значення втрат. У ідеальної моделі Log Loss дорівнює 0 (рисунок 3.3).

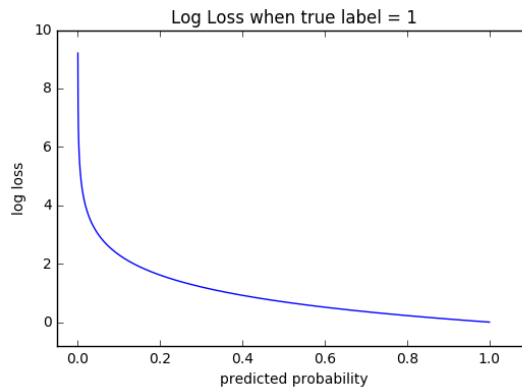


Рисунок 3.3 – Графік функції втрат Log Loss

Крос-ентропія та Log Loss трохи різняться залежно від контексту, але в машинному навчанні при обчисленні помилок від 0 до 1 вони зводяться до одного й того самого. Розберемо детальніше формулу:

N – кількість класів (собака, кіт, риба)

$\log$ – натуральний логарифм

y – двійковий індикатор (0 або 1), якщо мітка класу c є правильною класифікацією для спостереження $\mathbf{o}$

p – передбачення ймовірності, що спостереження $\mathbf{o}$ належить класу c
якщо $N = 2$ – це означає двійкову класифікацію

$$-\sum_i^N (y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

Якщо $N > 2$ (тобто багатокласова класифікація), ми розраховуємо окремі втрати для кожної мітки класу для кожного спостереження і підсумовуємо результат.

$$-\sum_{c=1}^N y_{o,c} \log(p_{o,c})$$

Для бінарної класифікації крос-ентропія дає:

$$CE(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1 - p) & \text{otherwise} \end{cases} \quad p_t = \begin{cases} p & \text{if } y = 1 \\ (1 - p) & \text{otherwise} \end{cases}$$

$$CE(p, y) = CE(p_t) = -\log(p_t).$$

Давайте побудуємо наведену вище шматочку функцію, щоб візуалізувати, що відбувається під нею (рисунок 3.4).



Рисунок 3.4 – Візуалізація функції крос-ентропії коли $y=1$ (зліва) та $y=0$ (праворуч)

Таким чином, функція бінарної крос-ентропії вловлює інтуїцію, що нейронна мережа повинна заплатити високий штраф (втрата $\rightarrow \infty$), коли оцінка ймовірності щодо мітки навчального прикладу повністю неправильна. З іншого боку, втрата повинна дорівнювати нулю (втрата = 0), коли оцінка ймовірності щодо мітки навчального прикладу вірна. Простіше кажучи, функція втрат повинна дорівнювати нулю лише у двох випадках:

1. Якщо приклад має позитивну мітку ($y = 1$), модель нейронної мережі має бути цілком упевнена, що приклад належить до позитивного класу, тобто $p = 1$.
2. якщо приклад має негативну мітку ($y = 0$), модель нейронної мережі має бути цілком упевнена, що приклад не належить до позитивного класу, тобто $p = 0$.

У нейронних мережах градієнт/похідна функції втрат визначає, чи слід збільшувати чи зменшувати ваги та зміщення нейронної мережі. Отже, давайте подивимося, як виглядає похідна від функції втрат двійкової крос-ентропії:

$$\frac{\partial L(y, p)}{\partial p} = \frac{\partial}{\partial p} (-y \log p - (1 - y) \log(1 - p)) = \dots = \frac{-y}{p} + \frac{1 - y}{1 - p}$$

Ми також можемо розбити похідну на кускові функції та візуалізувати її ефекти (рисунок 3.5):

$$\frac{\partial L(y, p)}{\partial p} = \begin{cases} \frac{-1}{p} & \text{якщо } y = 1 \\ \frac{1}{1 - p} & \text{якщо } y = 0 \end{cases}$$



Рисунок 3.5 – Похідна бінарної крос-ентропії коли $y=1$ (зліва) та $y=0$ (праворуч)

Слід відмітити, що коли $y=1$ та $p=0.5$ градієнт $= -2$; коли $y=0$ та $p=0.5$ градієнт $= 2$.

Позитивна похідна означатиме зменшення ваг, а негативна – збільшення ваг. Чим крутіше нахил (градієнт), тим невірнішим був прогноз.

- Якщо градієнт негативний, це означатиме, що ми дивимося на першу криву втрат (рисунок 3.5 зліва), де фактична мітка для прикладу є позитивною ($y = 1$). Єдиний спосіб довести втрати до нуля – це рухатися у напрямку, протилежному нахилу (градієнту), від негативного до позитивного. Отже, нам потрібно збільшити ваги та зміщення так, щоб $z = w_1 x_1 + w_2 x_2 + b > 0$, і, у свою чергу, оцінка ймовірності приналежності до позитивного класу дорівнює $p \approx \sigma(z) \approx 1$.

- Так само, коли градієнт позитивний, ми дивимося на другу криву втрат (рисунок 3.5 праворуч), де фактична мітка для прикладу негативна ($y = 0$). Єдиний спосіб довести втрати до нуля – це знову рухатись у напрямку, протилежному нахилу (градієнту), цього разу від позитивного до негативного. У цьому випадку нам потрібно буде зменшити ваги та зміщення так, щоб $z = w_1x_1 + w_2x_2 + b < 0$ і, отже, оцінена ймовірність приналежності до позитивного класу $p \approx \sigma(z) \approx 0$.

Докладніше про логістичну функцію помилки.

Цю функцію називають також «логлос» (logloss) і, як ми вже знаємо, перехресною або крос-ентропією (Cross Entropy) і часто використовують у завданнях класифікації. Розберемося докладніше, чому її використовують і який сенс вона має.

Згадаймо, як вирішується завдання лінійної регресії. Отже, ми хочемо отримати лінійну функцію (тобто ваги w), яка наближає цільове значення з точністю до помилки:

$$y = w^T x + \varepsilon, \varepsilon \sim \text{norm}(0, \sigma^2)$$

Тут ми припустили, що помилка нормально розподілена, x – ознаковий опис об'єкта (можливо, у ньому є і фіктивна константна ознака, щоб у лінійній функції був вільний член). Тоді ми знаємо як розподілені відповіді нашої функції і можемо записати функцію правдоподібності вибірки (тобто добуток щільностей, в які підставлені значення з навчальної вибірки) і скористатися методом максимальної правдоподібності (у якому для визначення значень параметрів береться максимум правдоподібності, а частіше його логарифму, детальніше про метод максимальної правдоподібності мова буде йти у наступних розділах):

$$p(y|x, w) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y - w^T x)^2}{2\sigma^2}\right]$$

$$L(w) = \log \prod_{i=1}^m p(y_i|x_i, w) = \sum_{i=1}^m \left[-\frac{1}{2} \log(2\pi\sigma^2) - \frac{(y_i - w^T x_i)^2}{2\sigma^2}\right] \rightarrow \max$$

$$\frac{1}{2\sigma^2} \sum_{i=1}^m (y_i - w^T x_i)^2 \rightarrow \min$$

Таким чином виявляється, що максимізація правдоподібності еквівалентна мінімізації середньоквадратичної помилки (MSE), тобто ця функція помилки не дарма широко використовується у завданнях регресії. Крім того, що вона цілком логічна, легко диференційована за параметрами та легко мінімізується, вона ще й теоретично обґрунтовується за допомогою методу максимальної правдоподібності у випадку, якщо лінійна модель відповідає даним з точністю до нормального шуму.

Для прикладу подивимося, як реалізується метод стохастичного градієнта (SGD) для мінімізації MSE: треба взяти похідну функції помилки для конкретного об'єкта і записати формулу корекції ваг у вигляді кроку у бік антиградієнта:

$$J(w) = \frac{1}{2} (w^T x - y)^2 \rightarrow \min$$

$$\frac{\partial J}{\partial w} = (w^T x - y)x$$

$$w := w - \alpha (w^T x - y)x$$

Отримали, що ваги лінійної моделі під час навчання методом стохастичного градієнтного спуску (SGD) коригуються з допомогою добавки вектора ознак. Коефіцієнт, з яким додають, залежить від «агресивності алгоритму» (параметр альфа, який називають темпом навчання) та різниці «відповідь алгоритму – правильна відповідь». До речі, якщо різниця нульова (тобто на даному об'єкті алгоритм видає точну відповідь), то корекція ваг не проводиться.

Тепер, нарешті, перейдемо безпосередньо до *«логлос»*. Розглянемо для прикладу завдання класифікації з двома класами: 0 і 1. Навчальну вибірку можна розглядати, як реалізацію узагальненої схеми Бернуллі: для кожного об'єкта генерується випадкова величина, яка з ймовірністю p (своїї для кожного об'єкта) набуває значення 1 і з ймовірністю $(1-p)$ – значення 0.

Припустимо, що ми якраз і будемо нашу модель так, щоб вона генерувала правильні ймовірності, але тоді можна записати функцію правдоподібності:

$a_i = a(x_i|w)$ – відповідь моделі, що залежить від w , на i – му об'єкті

$$p(y|X, w) = \prod_i p(y_i|x_i, w) = \prod_i a_i^{y_i} (1 - a_i)^{1-y_i} \rightarrow \max$$

$$\sum_i (-y_i \log a_i - (1 - y_i) \log(1 - a_i)) \rightarrow \min$$

Після логарифмування правдоподібності отримали, що його максимізація еквівалентна мінімізації останнього записаного виразу. Саме його і називають "*логістична функція помилки*". Для задачі бінарної класифікації, в якій алгоритм повинен видати ймовірність приналежності класу 1, вона логічна рівно настільки, наскільки логічна MSE у задачі лінійної регресії з нормальним шумом (оскільки обидві функції помилки виводяться з методу максимальної правдоподібності).

Налаштування на logloss

Один із методів «підганяння» відповідей алгоритму під logloss – калібрування Платта (Platt calibration). Ідея дуже проста. Нехай алгоритм породжує деякі оцінки приналежності до 1 класу – a . Метод спочатку розроблявся для калібрування відповідей алгоритму опорних векторів (SVM), цей алгоритм у найпростішій реалізації поділяє об'єкти гіперплощиною і просто видає номер класу 0 або 1, залежно від того, з якого боку гіперплощини об'єкт розташований. Але якщо ми побудували гіперплощину, то для будь-якого об'єкта можемо обчислити відстань до неї (зі знаком мінус, якщо об'єкт лежить у напівплощині нульового класу). Саме ці відстані зі знаком r ми перетворюватимемо на ймовірність за такою формулою:

$$a(x) = \text{sigmoid}(\alpha r(x) + \beta),$$

де невідомі параметри α , β зазвичай визначаються методом максимальної правдоподібності на відкладеній вибірці (calibration set).

Проілюструємо застосування методу на прикладі. На рисунку 3.6 показані відповіді (у вигляді ймовірностей) двох алгоритмів: градієнтного бустингу (lightgbm) та випадкового лісу (random forest).

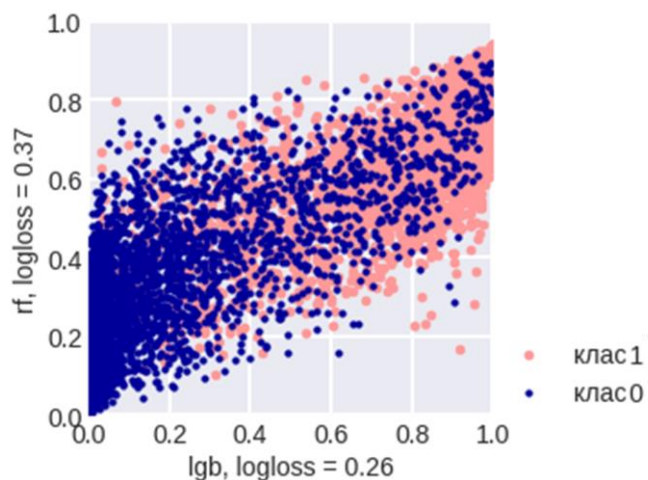


Рисунок 3.6 – Відповіді двох алгоритмів на всіх об'єктах вибірки.

Видно, що якість випадкового лісу набагато нижче і він досить обережний: занижує ймовірності у об'єктів класу 1 і завищує у об'єктів класу 0. Упорядкуємо всі об'єкти за зростанням ймовірностей (RF), розіб'ємо на k рівних частин і для кожної частини обчислимо середню відповідь алгоритму і середнє всіх правильних відповідей. Результат показано на рисунку 3.7 – точки зображені якраз у цих двох координатах.

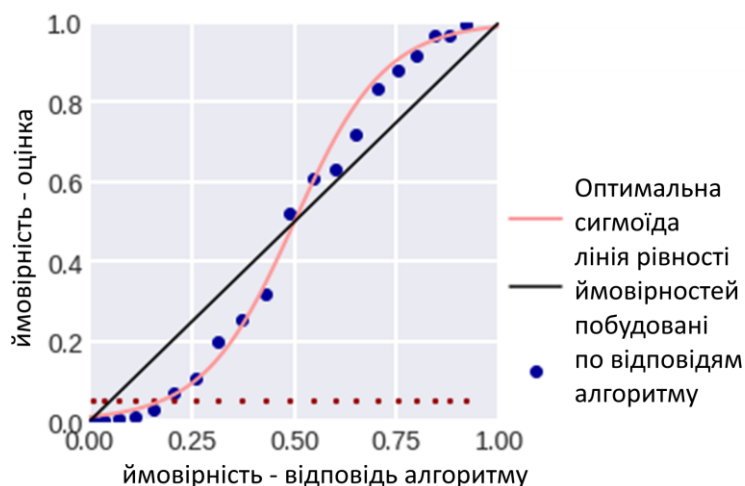


Рисунок 3.7 – Співвідношення ймовірностей: оціненої алгоритмом та усередненням.

Неважко бачити, що точки розташовуються на лінії, схожій на сигмоїду – можна оцінити параметр стиснення-розтягування у ній (рисунок 3.8). Оптимальна сигмоїда показана рожевим кольором на рисунку 3.7. Якщо відповідати такій сигмоїдній деформації, то логлос-помилка випадкового лісу знижується з 0.37 до 0.33.

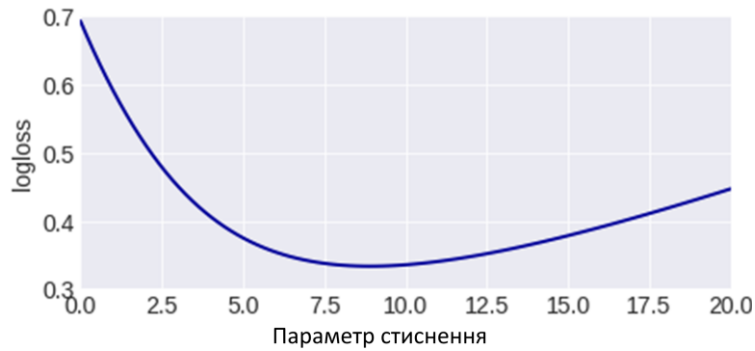


Рисунок 3.8 – Помилка в залежності від коефіцієнта стиснення аргументу сигмоїди.

Звернімо увагу, що тут ми деформували відповіді випадкового лісу (це були оцінки ймовірності – і всі вони лежали на відрізку $[0, 1]$), але з рисунка 3.8 видно, що для деформації потрібна саме сигмоїда. Практика показує, що у 80% ситуацій для покращення logloss-помилки треба деформувати відповіді саме за допомогою сигмоїди.

Ще один варіант калібрування – монотонна регресія (Isotonic regression).

Багатокласовий logloss

Для повноти картини зазначимо, що logloss узагальнюється і у разі кількох класів природним образом:

$$\text{logloss} = -\frac{1}{q} \sum_{i=1}^q \sum_{j=1}^l y_{ij} \log a_{ij}$$

тут q – це число елементів у вибірці, l – число класів, a_{ij} – це відповідь (ймовірність) алгоритму на i -му об'єкті на питання приналежності його до j -го класу, $y_{ij}=1$ якщо i -й об'єкт належить j -го класу, а в іншому випадку $y_{ij}=0$.

Розбіжність (дивергенція) Кульбака-Лейблера (KL-дивергенція, Kullback–Leibler divergence)

Розбіжність (дивергенцію) Кульбака-Лейблера (KL, Kullback–Leibler divergence) часто використовують (особливо в машинному навчанні, байєсівському підході та теорії інформації) для обчислення несхожості двох розподілів. Воно визначається за такою формулою:

$$D_{KL}(P|Q) = \int p(z) \log \frac{p(z)}{q(z)} dz,$$

де P і Q – розподіли (перше зазвичай «справжнє», а друге – те, про яке нам цікаво, наскільки воно схоже на справжнє), p і q – густини цих розподілів. Часто KL-розбіжність називають відстанню, хоча вона не є симетричною і не задовольняє нерівності трикутника. Для дискретних розподілів формулу записують так:

$$D_{KL}(P|Q) = \sum_i P_i \log \frac{P_i}{Q_i},$$

де P_i та Q_i – це ймовірності дискретних подій. Розглянемо конкретний об'єкт x з позначкою y . Якщо алгоритм видає ймовірність приналежності першому класу – a , то передбачуваний розподіл на подіях «клас 0», «клас 1» – $(1-a, a)$, а справжній – $(1-y, y)$, тому розбіжність Кульбака-Лейблера між ними

$$(1-y) \log \frac{(1-y)}{(1-a)} + y \log \frac{y}{a} = -(1-y) \log(1-a) - y \log a,$$

що точно збігається з logloss .

KL-дивергенція і крос-ентропія мають той самий ефект, що і цільові функції, що математично відрізняються на константу; логістичні втрати – це окремий випадок перехресної ентропії.

Hinge loss

Альтернативою крос-ентропії для задач двійкової класифікації є функція втрат Hinge loss, яка в основному розроблена для використання з моделями опорних векторів (SVM) (рисунок 3.9).

Вона призначена для використання з двійковою класифікацією, коли цільові значення перебувають у діапазоні $\{-1, 1\}$.

Функція втрати Hinge loss заохочує приклади мати правильний знак, присвоюючи більше помилок, коли є різниця у знаку між фактичним та прогнозованим значеннями класу.

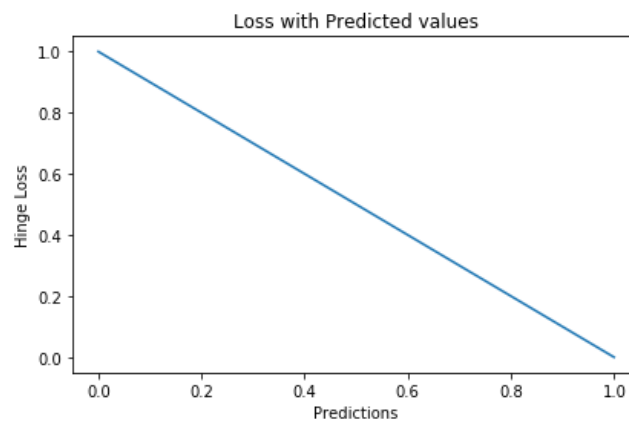


Рисунок 3.9 – Графік функції втрат Hinge loss

Квадратні втрати (Square loss)

Функція втрат Hinge loss має багато розширень, які часто досліджуються за допомогою моделей SVM.

Популярне розширення називається квадратичний Hinge loss (рисунок 3.10), який просто обчислює квадрат втрат Hinge loss. Це згладжує поверхню функції помилок і полегшує чисельну роботу з нею.

Якщо використання втрат Hinge loss дійсно призводить до кращої продуктивності для даної задачі двійкової класифікації, ймовірно, що квадрат втрат Hinge loss може бути придатною функцією.

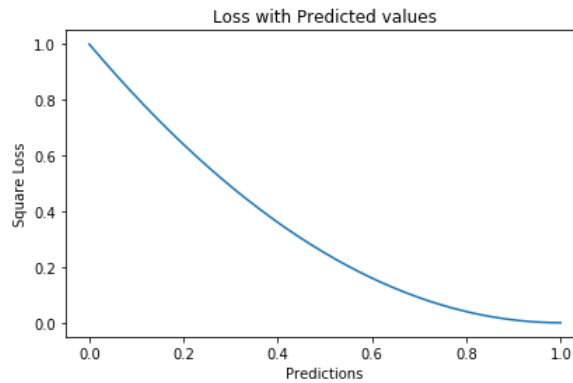


Рисунок 3.10 – Графік функції квадратичних втрат

Focal loss

Як приклад візьмемо двокласову класифікацію:

$$FL(p_t) = -\alpha_t(1 - p_t)^\gamma \log(p_t)$$

Focal loss – це поліпшення cross entropy loss ($p=p_t$), яка може регулювати вагу позитивних і негативних зразків (α_t) і вагу (γ) зразків, що важко класифікуються (рисунок 3.11). Коли α_t приймає значення 1, а γ приймає значення 0, формула є двокласовою функцією втрат перехресної ентропії, p_t – являє собою ймовірність передбачення.

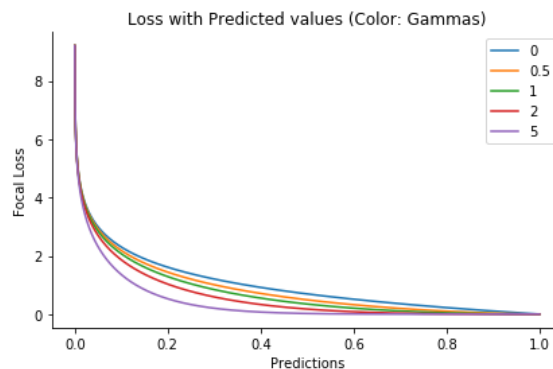


Рисунок 3.11 – Графік функції втрат Focal Loss

Коли вибірка розділена, p_t мала (наприклад, коли $y = 1$, p_t менше 0.5 класифікується неправильно), тому коефіцієнт модуляції γ прагне до 1, що мало відрізняється від вихідних втрат. Коли p_t прагне до 1 (коли класифікація правильна і зразок легко класифікується), коефіцієнт модуляції γ прагне до нуля, і внесок у загальні втрати невеликий.

Exponential loss

Експоненціальна втрата опукла і зростає експоненціально для негативних значень, що робить її більш чутливою до викидів (рисунок 3.12).

$$L(Y|f(X)) = \exp[-yf(x)]$$

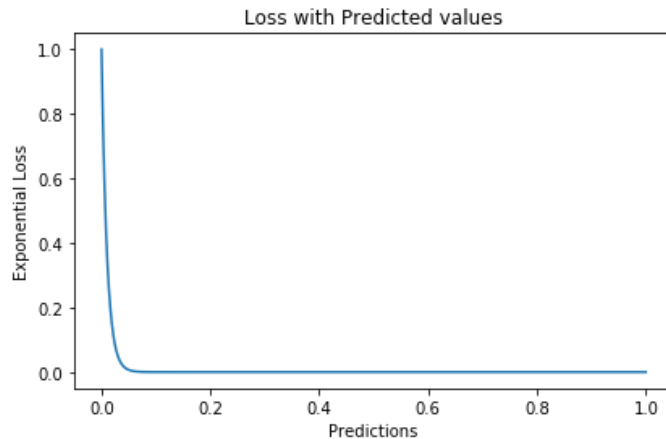


Рисунок 3.12 – Графік функції втрат Focal Loss

Типовим класифікатором, який використовує експоненційну функцію втрат, є алгоритм **AdaBoost**.

Архітектурні особливості нейронних мереж для задач регресії та класифікації

Розглянемо, які зазвичай використовуються вихідні шари та функції втрат у задачах машинного навчання (рисунок 3.13).

Завдання регресії.

Завдання, коли ви прогнозуєте дійсне число.

- Конфігурація вихідного рівня: один вузол із лінійною активацією.
- Функція втрат: наприклад, середня квадратична помилка (MSE).

Завдання бінарної класифікації

Завдання полягає в тому, щоб класифікувати приклад як один з двох класів. Або точніше, завдання сформульоване як передбачення ймовірності того, що приклад належить першому класу, наприклад, класу, якому ви надаєте ціле значення 1, тоді як іншому класу присвоюється значення 0.

- Конфігурація вихідного рівня: один вузол із сигмоподібною активаційною функцією.

- Функція втрат: наприклад, крос-ентропія, також звана логарифмічною функцією втрат.

Завдання мульти-класової класифікації

Це завдання полягає в тому, щоб класифікувати приклад як один з декількох класів. Завдання сформульоване як прогноз ймовірності того, що приклад належить кожному класу [15-18].

- Конфігурація вихідного рівня: один вузол кожного класу, що використовує функцію активації **softmax**.

- Функція втрат: наприклад, крос-ентропія, також звана логарифмічною функцією втрат.

Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Numerical value	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple labels, multiple classes	Sigmoid	Binary Cross Entropy

Рисунок 3.13 – Функції активації та функція втрат в залежності від типу задачі

Softmax. Функція активації багатокласової класифікації.

Softmax – це логістична функція для багатовимірного випадку. Цікаво, що україномовного терміну для цієї функції так і не з'явилося і в вітчизняній літературі вона так і зветься **softmax**.

Що таке "для багатовимірного випадку"? Це означає, що функцію застосовують не до окремого значення, а до вектора. Наприклад, її можна використовувати у тому випадку, коли стоїть завдання багатокласової класифікації. Для подібної класифікації мережу будують таким чином, що на останньому шарі кількість нейронів виявляється рівною кількості класів,

що шукаються. У цьому випадку кожен нейрон повинен видавати значення ймовірності приналежності об'єкта до класу, тобто значення між нулем і одиницею, проте нейрони у сумі повинні дати одиницю.

Ось так виглядає формула softmax:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{k=1}^K e^{z_k}}$$

де K – це кількість класів.

Для того щоб було зрозуміло, як працює ця формула, розглянемо простий приклад, в якому потрібно знайти приналежність об'єкта до одного з п'яти класів. Допустимо, на останньому шарі ми порахували z за формулою:

$$z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$$

і отримали такий вектор значень:

$$z^{[l]} = [1; -3; 2; 3; -1]$$

Спочатку порахуємо, знаменник формули, тобто підсумуємо всі елементи вектора:

$$[e^1; e^{-3}; e^2; e^3; e^{-1}] = [2.72; 0.05; 7.39; 20.09; 0.37]$$

значення при цьому виявиться приблизно рівним 30.62:

$$\sum_{k=1}^5 e^{z_k} = 30.62$$

Залишається кожне з значень вектора поділити на суму, що на виході дасть ось такий вектор:

$$[0.09; 0.001; 0.24; 0.66; 0.01]$$

У результаті, з ймовірністю 66% об'єкт із цього прикладу можна віднести до третього класу (якщо рахунок починати з нуля).

Для простоти сприйняття всі значення в прикладі округлені, тому в сумі виходить трохи більше одиниці.

Похідна softmax

Знову повернімося до визначення softmax

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} = \frac{e^{z_j}}{e^{z_1} + e^{z_2} + \dots + e^{z_n}}, j \in \{1, 2, \dots, n\}$$

Підрахунок похідної *Softmax* – одне з найпоширеніших завдань у машинному навчанні. Але тут є підводний камінь. Ми знаємо, що softmax обчислюється за одним із компонентів z_j вектору z . При обчисленні часткової похідної, компонента за якою беремо часткову похідну, НЕ обов'язково є компонентою, на якій обчислюється softmax. Або, говорячи математичною мовою, нам потрібно вирахувати $\partial\sigma(z_j)/\partial z_k$, де j та k не обов'язково рівні! Багато новачків приходять до розрахунку $\partial\sigma(z_j)/\partial z_j$, що є лише невеликою частиною проблеми.

Тепер порахуємо $\partial\sigma(z_j)/\partial z_k$. Ми будемо використовувати правило розрахунку часткової похідної:

$$(f(x)/g(x))' = (f'(x)g(x) - f(x)g'(x))/g(x)^2.$$

Також ми позначимо $\sum_{i=1}^n e^{z_i}$ як Σ для зручності.

1) Коли j дорівнює k маємо:

$$\begin{aligned} \frac{\partial\sigma(z_j)}{\partial z_k} &= \frac{\partial}{\partial z_j} \frac{e^{z_j}}{\Sigma} \dots \text{так як } j = k \\ &= \frac{\frac{\partial e^{z_j}}{\partial z_j} \Sigma - e^{z_j} \frac{\partial \Sigma}{\partial z_j}}{\Sigma^2} \dots \text{правило взяття похідної} \\ &= \frac{e^{z_j} \Sigma - e^{z_j} e^{z_j}}{\Sigma^2} = \frac{e^{z_j}}{\Sigma} - \left(\frac{e^{z_j}}{\Sigma}\right)^2 = \sigma(z_j) - (\sigma(z_j))^2 \\ &= \sigma(z_j)(1 - \sigma(z_j)). \end{aligned}$$

2) Коли j не дорівнює k маємо:

$$\begin{aligned}
\frac{\partial \sigma(z_j)}{\partial z_k} &= \frac{\partial}{\partial z_k} \frac{e^{z_j}}{\sum} = \frac{\frac{\partial e^{z_j}}{\partial z_k} \sum - e^{z_j} \frac{\partial \sum}{\partial z_k}}{\sum^2} \dots \text{правило взяття похідної} \\
&= \frac{0 \sum - e^{z_j} e^{z_k}}{\sum^2} \dots e^{z_j} \text{ константа коли рахуємо похідну по } z_k \\
&= -\frac{e^{z_j}}{\sum} - \frac{e^{z_k}}{\sum} = -\sigma(z_j)\sigma(z_k).
\end{aligned}$$

Тобто похідна softmax має вигляд:

$$\frac{\partial \sigma(z_j)}{\partial z_k} = \begin{cases} \sigma(z_j)(1 - \sigma(z_j)), & \text{коли } j = k \\ -\sigma(z_j)\sigma(z_k), & \text{коли } j \neq k \end{cases}$$

Крос-ентропія з Softmax

Іншим поширеним завданням машинного навчання є обчислення похідної крос-ентропії за допомогою softmax. Це можна записати так:

$$CE = \sum_{j=1}^n -y_i \log \sigma(z_j)$$

У задачі класифікації змінна n є кількість класів, а y_j є **one-hot** уявлення фактичного класу. **One-hot** – це вектор, у якому лише один компонент дорівнює 1, проте інші компоненти дорівнюють нулю. Іншими словами, є тільки одна 1 $y_1, y_2, \dots, y_n$ а всі інші рівні нулю (рисунок 3.14)

	<i>data</i>	<i>AI</i>	<i>book</i>	<i>Algorithms</i>
<i>data</i>	1	0	0	0
<i>AI</i>	0	1	0	0
<i>book</i>	0	0	1	0
<i>Algorithms</i>	0	0	0	1

Рисунок 3.14 – Приклад one-hot даних

Тобто, наприклад, для завдання класифікації з 5 класами, якщо конкретна точка даних має клас 4, її мітка класу $y = (0,0,0,1,0)$.

Тепер давайте обчислимо похідну крос-ентропію з softmax. Ми будемо використовувати ланцюгове правило взяття похідної складної

функції, яке вже розглядалося раніше у розділі зворотного поширення помилки:

$$\left(f(g(x))\right)' = f'(g(x))g'(x)$$

$$\begin{aligned} \frac{\partial}{\partial z_k} CE &= \frac{\partial}{\partial z_k} \sum_{j=1}^n (-y_j \log \sigma(z_j)) = \\ &= - \sum_{j=1}^n y_j \frac{\partial}{\partial z_k} \log \sigma(z_j) \dots \text{правило додавання, } -y_j \text{ це константа} = \\ &= - \sum_{j=1}^n y_j \frac{1}{\sigma(z_j)} \frac{\partial}{\partial z_k} \sigma(z_j) \dots \text{правило ланцюга} = \\ &= -y_k \frac{\sigma(z_k)(1 - \sigma(z_k))}{\sigma(z_k)} - \sum_{j \neq k} y_j \frac{-\sigma(z_j)\sigma(z_k)}{\sigma(z_j)} \dots \text{випадки: } j = k \text{ та } j \neq k \\ &= -y_k(1 - \sigma(z_k)) + \sum_{j \neq k} y_j \sigma(z_k) = -y_k + y_k \sigma(z_k) + \sum_{j \neq k} y_j \sigma(z_k) = \\ &= -y_k + \sigma(z_k) \sum_j y_j. \end{aligned}$$

Оскільки у – one-hot вектор, то $\sum_j y_j = 1$. Таким чином, похідна крос-ентропії з softmax остаточно буде

$$\frac{\partial}{\partial z_k} CE = -y_k + \sigma(z_k)$$

Це дуже просте рівняння, що легко обчислюється.

ТЕМА №4: РЕГУЛЯРИЗАЦІЯ ФУНКЦІЙ ВТРАТ. НОРМАЛІЗАЦІЯ ДАНИХ. ВАСН - НОРМАЛІЗАЦІЯ

Регуляризація

Припустимо, що ми розробляємо модель машинного навчання. Модель зазвичай навчається з використанням набору тренувальних даних і тестується на тестовому наборі даних. Точність моделі зазвичай знижується, якщо набір даних не було попередньо оброблено. Фактично, більшість часу йде на попередню обробку даних перед навчанням моделі.

Які можуть бути причини низької точності моделі?

Є дві основні причини, про які коротко вже йшла мова у попередніх розділах [5-8]:

1. Underfitting
2. Overfitting

Underfitting (недонавченість)

Underfitting – це явище, у якому модель не вивчила узагальнену модель набору даних. Це може бути через неадекватний набір даних, або через використання невідповідних алгоритмів машинного навчання для цього конкретного набору даних.

Розглянемо приклад (рисунок 4.1). Графік містить дані точок, які представляють синусоїдальну криву. Ці дані було введено в алгоритм лінійної регресії. Лінія – це прогнозований результат моделі. Зрозуміло, що точність моделі не надто велика! Чому так? Оскільки лінійну модель було навчено на нелінійному наборі даних. Отже, оптимізація лінійної лінії до нелінійних даних робить модель неточною. Середньоквадратична помилка дорівнює 0.26704 умовних одиниць. Очевидно, що якби замість лінії ми використовували криву третього порядку, то результат був би набагато кращим. І насправді за допомогою лінійної регресії ми можемо навчати нелінійні моделі.

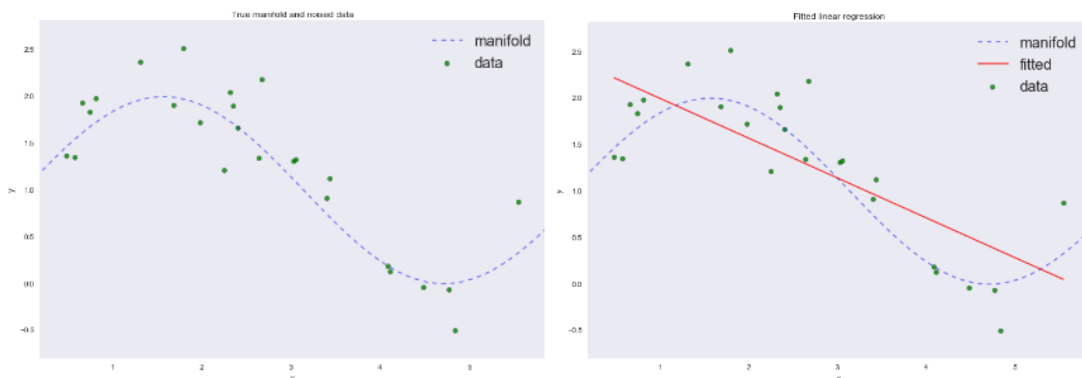


Рисунок 4.1 – Приклад недонавченої моделі (underfitting)

Overfitting (перенавчання)

Перенавчання – це явище, при якому модель дуже багато чого навчилася з тренувального набору даних. Перенавчена модель дійсно добре працює з тренувальними даними, на яких вона була навчена, але її точність має тенденцію до зниження при використанні з новим набором даних (тестовим).

Розглянемо той самий приклад. І тут скористаємося моделлю поліноміальної регресії (рисунок 4.2). У лінійній регресії ми обмежували простір гіпотез лише лінійними функціями від ознак. Давайте тепер розширимо простір гіпотез до всіх поліномів ступеня p .

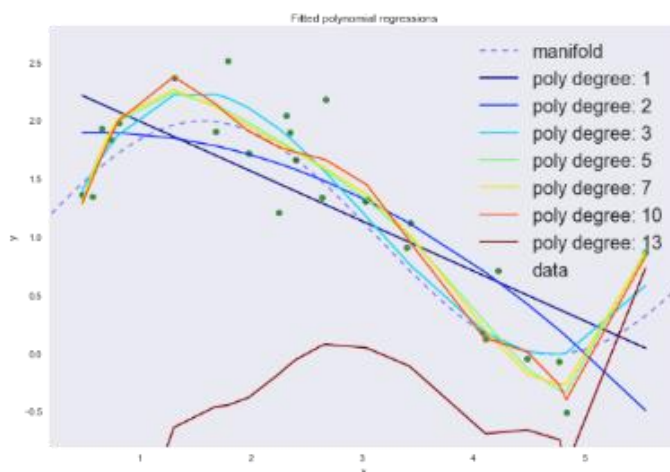


Рисунок 4.2– Приклад перенавченої моделі (overfitting)

На рисунку 4.2 ми можемо спостерігати одразу два феномени. Поки не звертаємо увагу на 13-у ступінь полінома. При збільшенні ступеня полінома середня помилка продовжує зменшуватися, хоча ми начебто були

впевнені, що саме кубічний поліном повинен найкраще описувати наші дані.

p	error
1	0.26704
2	0.22495
3	0.08217
5	0.05862
7	0.05749
10	0.0532
13	5.76155

Це явна ознака перенавчання, яку можна помітити з візуалізації навіть не використовуючи тестовий набір даних: при збільшенні ступеня полінома вище третьої модель починає інтерполювати дані замість екстраполяції. Іншими словами, графік функції проходить точно через точки з тренувального набору даних, причому чим вищий ступінь полінома, тим через більшу кількість точок він проходить. Ступінь полінома відбиває складність моделі. Таким чином, складні моделі, у яких ступенів свободи досить багата, можуть запам'ятати весь тренувальний набір, повністю втрачаючи узагальнюючу здатність. Це і є проявом негативної сторони принципу мінімізації емпіричного ризику.

Щодо полінома 13-го ступеня – тут явна помилка, оскільки, викликана роботою чисельного процесора NumPy у Python і пов'язана з значеннями, що розходяться при обчисленні коефіцієнтів лінійної регресії. Але це зовсім інша історія, не вдаватимемося в подробиці.

Амплітуда коефіцієнтів також змінилася, хоча скакати у різні боки вони не перестали. Пам'ятаємо, що поліном третього ступеня повинен найкраще описувати наші дані, хотілося б, щоб у результаті регуляризації всі коефіцієнти при поліноміальних ознаках ступеня вище третього дорівнювали нулю. І, виявляється, є і такий регуляризатор.

Регуляризація запобігає перенавченню!

Методи регуляризації використовуються для зменшення похибки шляхом відповідної підгонки функції до цього тренувального набору, щоб уникнути перенавчання. Ці функції суттєво зменшують коефіцієнти (β) кожної ознаки, тим самим зменшуючи шанси на відміну значень. Розглянемо для визначеності рівняння лінійної регресії:

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i$$

Типи регуляризації

1. L1
2. L2
3. Elastic net regularization
4. Dropout (тільки для нейронних мереж)

L1 (Регуляризація Lasso):

Ідея, що лежить в основі регуляризації L1, полягає в тому, щоб зводити набір даних тільки до найважливіших ознак, які впливатимуть на цільову змінну. Регуляризація L1 додає штраф до функції втрат, що дорівнює сумі абсолютних значень коефіцієнтів.

$$Error_{L1} = Error(Cost Function) + \lambda \sum_{i=0}^N |\beta_i|$$

де β_i – це параметри. Тоді для прикладу MSE втрат отримуємо

$$\sum_{i=1}^N (Y_i - \sum_{j=1}^p X_{ij} \beta_j)^2 + \lambda \sum_{j=1}^p |\beta_j|$$

Додавши вищезгаданий штраф, деякі з коефіцієнтів стають 0 та інші ознаки будуть найкориснішими. Цей метод регуляризації можна розглядати, як метод вибору ознак.

L2 (Регуляризація Ridge):

Регуляризація L2 додає штраф, що дорівнює сумі квадратів значень коефіцієнтів.

$$Error_{L2} = Error (Cost Function) + \lambda \sum_{i=0}^N \beta_i^2$$

$$\sum_{i=1}^N (Y_i - \sum_{j=1}^p X_{ij} \beta_j)^2 + \lambda \sum_{j=1}^p \beta_j^2$$

λ у рівняннях вище є гіперпараметром, що контролює інтенсивність штрафу.

Коли $\lambda \rightarrow 0$, результати аналогічні до лінійної регресії.

Коли $\lambda \rightarrow \infty$, всі ознаки зменшено до 0.

Оскільки застосовується штраф, коефіцієнти не зазнають значних змін до 0, швидше вони повільно зменшуються до 0. Отже, L2 не може використовуватися для вибору ознак на відміну від L1.

В обох випадках чим більший штраф, тим меншими стають коефіцієнти.

Elastic net regularization

Ця регуляризація є комбінацією регуляризації L1 та L2.

Покарання застосовується так (по суті маємо експоненційну складову):

$$Error_{L1L2} = Error + \lambda((1 - \alpha) \sum_{i=0}^N \beta_i^2 + \alpha \sum_{i=0}^N |\beta_i|)$$

У цьому випадку α є загальним параметром, який встановлює співвідношення між L1 та L2. Таким чином, результатом буде гібридом регуляризації L1 та L2.

На рисунку 4.3 надане геометричне подання розглянутих регуляризацій.

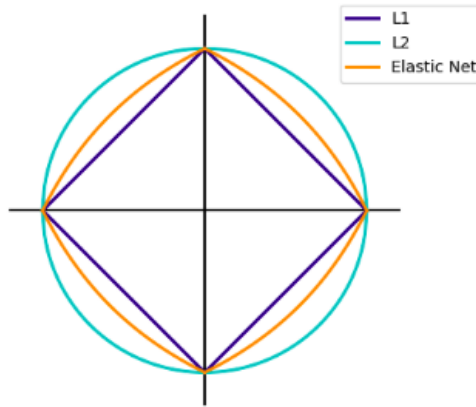


Рисунок 4.3 – Геометричне представлення компонент регуляризації

Які плюси та мінуси кожної регуляризації L1/L2?

Регуляризація L1 може вирішити проблему мультиколінеарності, обмежуючи норму коефіцієнта і закріплюючи деякі значення коефіцієнта до 0. У обчислювальному відношенні регресія Лассо (регресія зі штрафом L1) є квадратичною програмою, на вирішення якої потрібні деякі спеціальні інструменти. Регуляризація L1 іноді використовується як метод вибору ознак. Припустимо, у вас є якесь жорстке обмеження на кількість ознак, які ви можете використовувати (бо збір даних для всіх ознак недешевий, або у вас жорсткі інженерні обмеження на кількість значень, які можна зберігати, і т. д.). Ви можете спробувати налаштувати штраф L1, щоб досягти бажаної кількості ненульових ознак.

Регуляризація L2 може вирішити проблему мультиколінеарності, обмежуючи норму коефіцієнта та зберігаючи всі змінні. Малоймовірно, що коефіцієнт точно дорівнює 0. Це не обов'язково є недоліком, якщо з якоїсь причини не важливий вектор розріджених коефіцієнтів.

У разі регресії це «класичне» вирішення проблеми оцінки регресії з більшою кількістю ознак, ніж спостережень. Регуляризація L2 може оцінити коефіцієнт кожної ознаки, навіть якщо є більше ознак, ніж спостережень (насправді – це і було початковою мотивацією).

Як альтернатива elastic net regularization допускає регуляризацію L1 і L2 як особливі випадки. Типовий приклад використання даних у галузі

для фахівців з обробки даних полягає в тому, що ви просто хочете вибрати найкращу модель, але вам не обов'язково байдуже, чи буде вона покарана з використанням L1, L2 або обох.

Застосування норм L1 чи L2 для накладення штрафів на великі коефіцієнти є поширеним способом регуляризації лінійної чи логістичної регресії. Однак багато фахівців недооцінюють важливість стандартизації значень ознак перед застосуванням регуляризації.

Звернімо увагу, що дуже важливо правильно вибрати λ . Якщо коефіцієнт занадто малий, то ефект від регуляризації буде мізерний, якщо ж занадто великий – модель обнуляє всі ваги.

Dropout - метод вирішення проблеми перенавчання в нейронних мережах.

Графічне подання методу Dropout наведено на рисунку 4.4, взяте зі статті, де він вперше був представлений [24]. Ліворуч – нейронна мережа до того, як до неї застосували Dropout, справа – та сама мережа після Dropout. Мережа, зображена зліва, використовується під час тестування, після навчання параметрам.

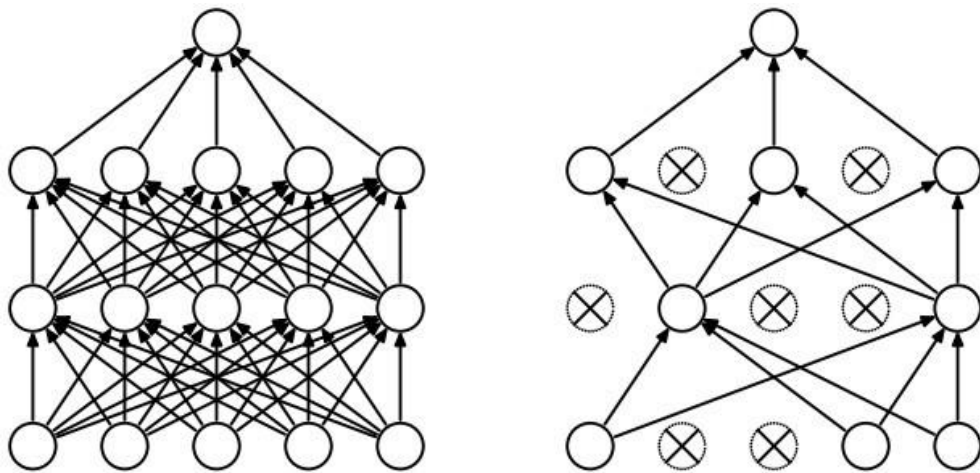


Рисунок 4.4 – Стандартна нейронна мережа (зліва) та мережа після використання dropout (праворуч)

Головна ідея Dropout – замість навчання однієї нейронної мережі навчити ансамбль кількох нейронних мереж, а потім усереднити отримані результати [24].

Мережі для навчання виходять за допомогою виключення з мережі (dropping out) нейронів з ймовірністю p , таким чином, ймовірність того, що нейрон залишиться в мережі, становить $q=1-p$. "Виключення" нейрона означає, що при будь-яких вхідних даних або параметрах він повертає 0.

Виключені нейрони не роблять свій внесок у процес навчання на жодному з етапів алгоритму зворотного поширення помилки (backpropagation); тому виключення хоча б одного з нейронів рівносильне навчанню нової нейронної мережі.

У стандартній нейронній мережі похідна, отримана кожним параметром, повідомляє йому, як він повинен змінитися, щоб, враховуючи діяльність інших блоків, мінімізувати функцію кінцевих втрат. Тому блоки можуть змінюватися, виправляючи помилки інших блоків. Це може призвести до надмірної спільної адаптації (co-adaptation), що, у свою чергу, призводить до перенавчання, оскільки ці спільні адаптації неможливо узагальнити на дані, які не брали участь у навчанні. Автори статті про «дропаут» висунули гіпотезу, що Dropout запобігає сумісній адаптації для кожного прихованого блоку, роблячи присутність інших прихованих блоків ненадійною. Тому прихований блок не може покладатися на інші блоки у виправленні помилок.

У двох словах, Dropout добре працює на практиці, тому що запобігає взаємoadаптації нейронів на етапі навчання. Отримавши приблизне уявлення про метод Dropout, розглянемо його докладніше.

Як працює Dropout

Як говорилося, Dropout вимикає нейрони з ймовірністю p і, як наслідок, залишає їх включеними з ймовірністю $q=1-p$.

Ймовірність вимикання кожного нейрона однакова. Це означає наступне:

За умови, що

- $h(x)=xW+b$ – це лінійна проекція вхідного d_i -вимірному вектора x на d_h -вимірний простір вихідних значень;
- $a(h)$ – це функція активації,

застосування Dropout до даної проекції на етапі навчання можна подати як змінену функцію активації:

$$f(h)=D \odot a(h),$$

де $D = (X_1, \dots, X_{d_h})$ – d_h -вимірний вектор випадкових величин X_i , розподілених за законом Бернуллі.

X_i має такий розподіл ймовірностей:

$$f(k; p) = \begin{cases} p & \text{якщо } k = 1 \\ 1 - p & \text{якщо } k = 0 \end{cases}$$

де k – це усі можливі вихідні значення.

Очевидно, що ця випадкова величина ідеально відповідає Dropout, застосованому до одного нейрона. Справді, нейрон вимикають із ймовірністю $p=P(k=1)$, інакше – залишають включеним.

Подивимось застосування Dropout до i -го нейрону:

$$O_i = X_i a\left(\sum_{k=1}^{d_i} w_k x_k + b\right) = \begin{cases} a\left(\sum_{k=1}^{d_i} w_k x_k + b\right), & \text{якщо } X_i = 1 \\ 0, & \text{якщо } X_i = 0 \end{cases}$$

де $P(X_i = 0) = p$.

Так як на етапі навчання нейрон залишається в мережі (не піддається вимкненню) з ймовірністю q , на етапі тестування нам необхідно емулювати поведінку ансамблю нейронних мереж, використаного під час навчання. Для цього автори пропонують на етапі тестування помножити функцію активації на коефіцієнт q . Таким чином,

$$\text{Під час навчання } O_i = X_i a\left(\sum_{k=1}^{d_i} w_k x_k + b\right)$$

$$\text{Під час тестування } O_i = q a\left(\sum_{k=1}^{d_i} w_k x_k + b\right)$$

Зворотній (Inverted) Dropout

Можливо використовувати трохи інший підхід – зворотний Dropout. У цьому разі ми множимо функцію активації на коефіцієнт не під час тестового етапу, а під час навчання.

Коефіцієнт дорівнює зворотній величині ймовірності того, що нейрон залишиться в мережі: $1/1 - p = 1/q$ таким чином,

$$\text{Під час навчання } O_i = 1/q X_i a\left(\sum_{k=1}^{d_i} w_k x_k + b\right)$$

$$\text{Під час тестування } O_i = a\left(\sum_{k=1}^{d_i} w_k x_k + b\right)$$

У багатьох фреймворках для глибокого навчання Dropout реалізований якраз у цій модифікації, тому що при цьому необхідно лише одного разу описати модель, а потім запускати навчання і тестування на цій моделі, змінюючи лише параметр (коефіцієнт Dropout).

У разі прямого Dropout ми змушені змінювати нейронну мережу для проведення тестування, так як без множення на q нейрон повертатиме значення вищі, ніж ті, які очікують отримати наступні нейрони. Саме тому реалізація зворотного Dropout зустрічається частіше.

Dropout безлічі нейронів

Легко помітити, що шар h з n нейронів на окремому етапі навчання можна розглядати як ансамбль з n експериментів Бернуллі з ймовірністю успіху p .

Таким чином, на виході шару h ми отримуємо таку кількість виключених нейронів:

$$Y = \sum_{i=1}^{d_h} (1 - X_i)$$

Так як кожен нейрон представлений у вигляді випадкової величини, розподіленої за законом Бернуллі, і всі ці величини незалежні, загальна кількість виключених нейронів також випадкова величина, але має біномний розподіл:

$$Y \sim Bi(d_h, p),$$

де ймовірність k успішних подій за n спроб характеризується наступною щільністю розподілу:

$$f(k; n, p) = \binom{n}{k} p^k (1 - p)^{n-k}$$

Цю формулу легко пояснити так:

- $p^k (1 - p)^{n-k}$ – це ймовірність отримання послідовності k успішних подій за n спроб, і, отже, $n-k$ неуспішних.
- $\binom{n}{k}$ – це біноміальний коефіцієнт, що використовується для розрахунку кількості можливих успішних послідовностей.

Тепер ми можемо використовувати цей розподіл, щоб розрахувати можливість відключення певної кількості нейронів.

Використовуючи метод Dropout, ми фіксуємо коефіцієнт Dropout p для певного шару та очікуємо, що з цього шару буде виключено пропорційну кількість нейронів.

Наприклад, якщо шар, до якого ми застосували Dropout, складається з $n=1024$ нейронів, а $p=0.5$ ми очікуємо, що 512 з них буде відключено. Давайте перевіримо це твердження:

$$Y = \sum_{i=1}^{1024} X_i \sim Bi(1024, 0.5)$$

$$P(Y = 512) = \binom{1024}{512} 0.5^{512} (1 - 0.5)^{1024-512} \approx 0.025$$

Як бачимо, ймовірність відключення рівно $n_p=512$ нейронів становить лише 0.025!

Dropout та інші регуляризатори

Dropout часто використовується з L2-нормалізацією та іншими методами обмеження параметрів (наприклад, стандартизацією та/або нормалізацією даних). Методи нормалізації допомагають підтримувати низькі значення параметрів моделі.

Dropout сам по собі не може запобігти зростанню значень параметрів на етапі оновлення. Більш того, зворотний Dropout веде до того, щоб кроки оновлення стали навіть більшими.

Таким чином.

1. Dropout існує у двох модифікаціях: прямий (використовується нечасто) та зворотний.
2. Dropout на окремому нейроні може бути представлений як випадкова величина із розподілом Бернуллі.
3. Dropout на багатьох нейронах може бути представлений як випадкова величина з біноміальним розподілом.
4. Незважаючи на те, що ймовірність того, що з мережі буде вимкнено рівно p нейронів, p – середня кількість нейронів, відключених у шарі з n нейронів.
5. Зворотний Dropout підвищує швидкість навчання.
6. Зворотний Dropout слід використовувати разом з іншими методами нормалізації, що обмежують значення параметрів, щоб спростити процес вибору швидкості навчання.
7. Dropout допомагає запобігти проблемі перенавчання у глибоких нейронних мережах.

Нормалізація даних, передобробка.

Навіщо нормалізувати?

Багато алгоритмів машинного навчання намагаються знайти тенденції у даних шляхом порівняння характеристик точок даних. Однак виникає проблема, коли ознаки сильно різняться за масштабами [4-5].

Наприклад, розглянемо набір даних про будинки. Двома потенційними характеристиками можуть бути кількість кімнат у будинку та загальний вік будинку у роках. Алгоритм машинного навчання може спробувати передбачити, який будинок підійде вам найкраще. Проте, коли

алгоритм порівнює точки даних, об'єкт із великим масштабом повністю домінує над іншим (рисунок 4.5).

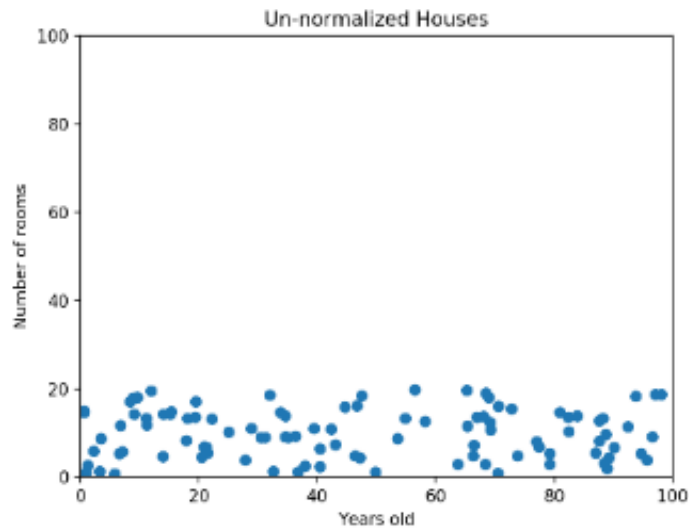


Рисунок 4.5 – Приклад ознак у наборі даних про будинки

Коли дані виглядають так стисло, ми знаємо, що ми маємо проблему. Алгоритм машинного навчання повинен розуміти, що існує величезна різниця між будинком з 2 кімнатами та будинком з 20 кімнатами. Але прямо зараз, оскільки два будинки можуть знаходитись на відстані 100 років один від одного, різниця в кількості кімнат менше впливає на загальну різницю.

Як крайній приклад уявіть, як би виглядав графік, якби по осі абсцис була вартість будинку. Дані будуть виглядати ще стислішими; різниця в кількості кімнат буде ще менш актуальною, тому що вартість двох будинків може відрізнятись у тисячі доларів.

Таким чином, мета нормалізації – зробити всі точки даних однаковими за масштабом, щоб кожна ознака була однаково важлива. На рисунку 4.6 показані ті самі домашні дані, нормалізовані з використанням мінімальної та максимальної нормалізації.

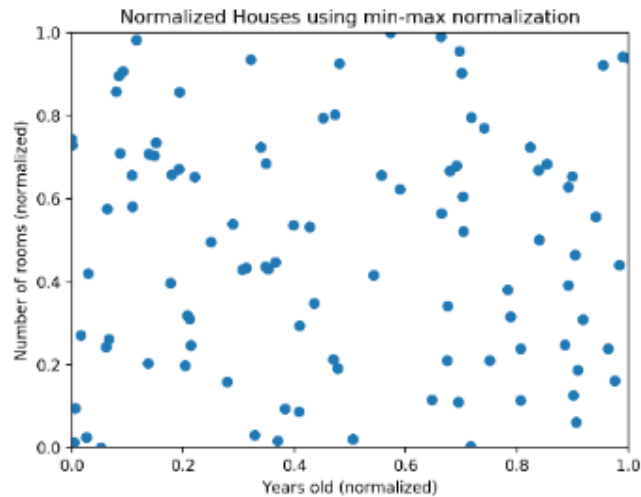


Рисунок 4.6 – Приклад даних з рисунку 4.6 після нормалізації

Алгоритми на основі градієнтного спуску

Алгоритми машинного навчання, такі як лінійна регресія, логістична регресія, нейронні мережі тощо, які використовують градієнтний спуск як метод оптимізації, вимагають масштабування даних. Поглянемо на формулу градієнтного спуску:

$$\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)} - y^{(i)})) x_j^{(i)}$$

Наявність значення X у формулі вплине на розмір кроку градієнтного спуску. Різниця в діапазонах функцій призведе до різних розмірів кроків для кожної функції. Щоб гарантувати, що градієнтний спуск плавно рухається до мінімумів і що кроки градієнтного спуску оновлюються однаково для всіх функцій, ми і масштабуємо дані перед їх передачею в модель (рисунок 4.7).

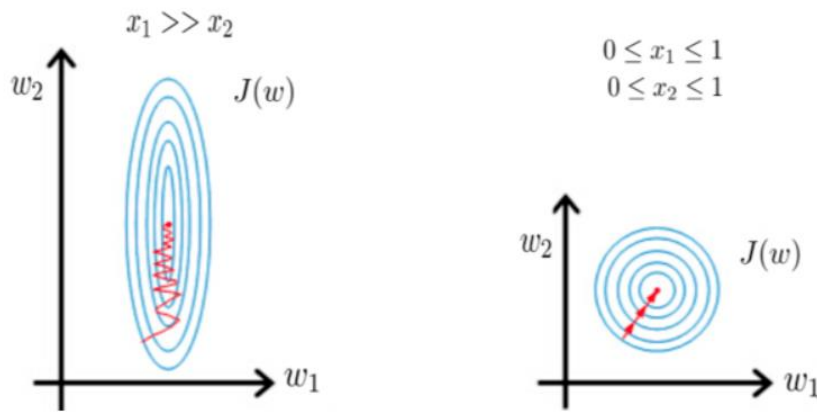


Рисунок 4.7 – Градієнтний спуск без нормалізації (зліва) та із нормалізацією (справа)

Тобто, наявність функцій аналогічного масштабу може допомогти градієнтному спуску швидше сходиться до мінімуму.

Нормалізація min - max

Нормалізація мінімум-максимум – один із найпоширеніших способів нормалізації даних. Для кожної ознаки мінімальне значення цієї ознаки перетворюється на 0, максимальне значення перетворюється на 1, проте інші значення перетворюються на десяткове число від 0 до 1.

Наприклад, якщо мінімальне значення об'єкта було 20, а максимальне значення 40, 30 буде перетворено приблизно в 0.5, оскільки воно знаходиться на півдорозі між 20 і 40. Формула виглядає наступним чином:

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Нормалізація *min-max* має досить істотний недолік: вона дуже добре обробляє викиди. Наприклад, якщо у нас 99 значень від 0 до 40, а одне значення дорівнює 100, тоді всі 99 значень буде перетворено на значення від 0 до 0.4. Ці дані так само стиснуті, як і раніше! (рисунок 4.8).

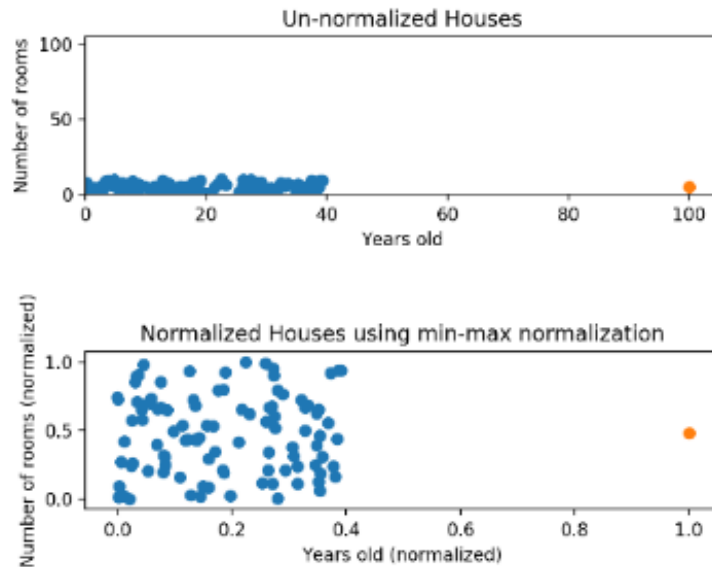


Рисунок 4.8 – Приклад використання нормалізації min-max

Нормалізація усунула проблему стиснення по осі Y, але вісь X, як і раніше, проблематична. Тепер, якби ми порівняли ці точки, вісь Y переважала; вісь y може відрізнятись на 1, а вісь x може відрізнятись лише на 0.4.

Z- Score нормалізація

Нормалізація Z-score – це стратегія нормалізації даних, що дозволяє уникнути проблеми з викидами. Формула нормалізації Z-score наведено нижче:

$$x' = \frac{x - x_{mean}}{\sigma}, z = \frac{x - \mu}{\sigma}$$

Тут μ – це середнє значення ознаки, а σ – це стандартне відхилення ознаки. Якщо значення точно дорівнює середньому значенню всіх значень ознаки, воно буде нормалізовано до 0. Якщо воно нижче за середнє, це буде негативне число, а якщо воно вище за середнє – позитивне. число. Розмір цих негативних та позитивних чисел визначається стандартним відхиленням вихідного об'єкта. Якщо у ненормалізованих даних було велике стандартне відхилення, нормалізовані значення будуть ближчими до 0.

Подивимося на ті самі дані, що й раніше, але цього разу використовувалася нормалізація z- score (рисунок 4.9) .

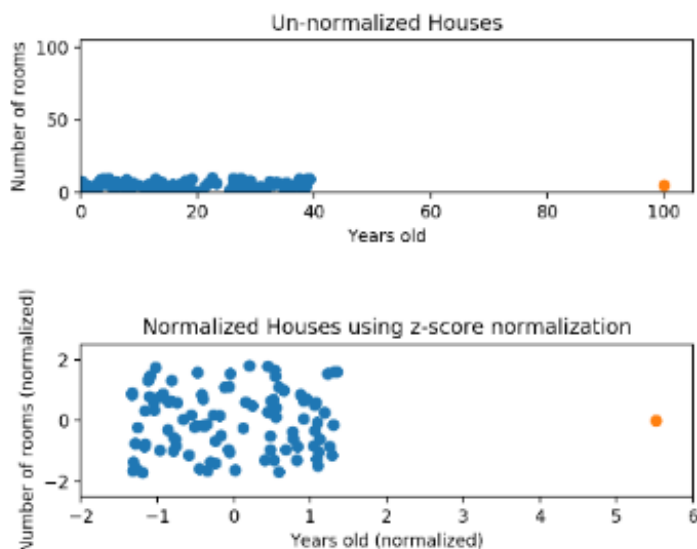


Рисунок 4.9 – Приклад використання нормалізації z-score

Хоча дані, як і раніше , виглядають стислими, зверніть увагу, що точки тепер мають приблизно однаковий масштаб для обох функцій – майже всі точки знаходяться між -2 і 2 як по осі x, так і по осі y. Єдиний потенційний недолік у тому, що функції не на точному масштабі.

З нормалізацією min-max ми гарантовано змінили форму обох наших функцій, щоб вони були між 0 і 1. Використовуючи нормалізацію z-score, вісь x тепер має діапазон від -1,5 до 1,5, а вісь y має діапазон приблизно від - 2 до 2. Це, безумовно, краще ніж раніше; вісь x, яка раніше мала діапазон від 0 до 40 більше не домінує над віссю y.

Висновок

Нормалізація даних – важлива частина машинного навчання. Ви можете мати чудовий набір даних з багатьма чудовими функціями, але якщо ви забудете нормалізувати дані, одна з цих функцій може повністю домінувати над іншими. Начебто ви викидаєте майже всю свою інформацію! Нормалізація вирішує цю проблему.

- Нормалізація мінімум-максимум: гарантує, що всі функції матимуть однаковий масштаб, але погано справляється з викидами.
- Нормалізація Z-score: обробляє викиди, але не виробляє нормалізовані дані з такою самою шкалою.

Додаткові нормалізації

Середня нормалізація – полягає в тому, щоб змінити спостереження так, щоб їх можна було описати як нормальний розподіл. Нормальний розподіл (розподіл Гауса), також відомий як дзвоноподібна крива, являє собою конкретний статистичний розподіл, в якому спостерігаються приблизно рівні значення (рисунк 4.10). Дані падають вище та нижче середнього, варіюються від -1 до 1 із середнім значенням $= 0$.

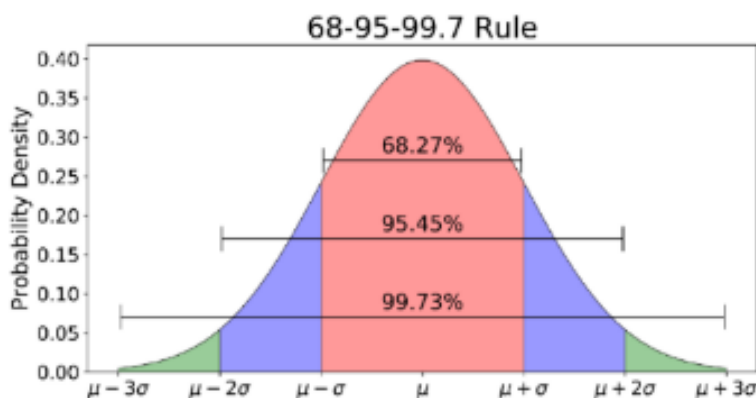


Рисунок 4.10 – Приклад нормального розподілу

Бінаризувати дані (зробити двійковими) – можна конвертувати дані, використовуючи двійковий поріг. Усі значення вище порогового значення позначені як 1, а всі рівні або нижче позначені як 0. Це називається бінаризацією даних або встановленням порога даних. Це може бути корисним, коли є ймовірність того, що ви хочете отримати чіткі значення. Це також корисно при розробці функцій, коли ви хочете додати нові функції, які вказують на щось важливе.

Остання нормалізація: **Одиничний вектор** – масштабування виконується з урахуванням того, що всі значення функції мають одиничну довжину. Працюючи з об'єктами з жорсткими кордонами це дуже корисно.

Наприклад, при роботі з даними зображення кольори можуть змінюватись від 0 до 255.

$$x' = \frac{x}{||x||}$$

Батч-нормалізація (batch normalization)

Батч-нормалізація – це спосіб прискорення глибокого навчання. Метод вирішує наступну проблему, що перешкоджає ефективному навчанню нейронних мереж: у міру поширення сигналу по мережі, навіть якщо ми нормалізували його на вході, пройшовши через внутрішні шари, він може сильно спотворитися як по маточікуванню, так і по дисперсії (дане явище називається *внутрішнім коваріаційним зсувом*), що загрожує серйозними невідповідностями між градієнтами на різних рівнях. Тому нам доводиться використовувати сильніші регуляризатори, уповільнюючи тим самим темп навчання [25].

Батч-нормалізація пропонує вельми просте вирішення даної проблеми: нормалізувати вхідні дані таким чином, щоб отримати нульове маточікування та одиничну дисперсію. Нормалізація виконується перед входом до кожного шару. Це означає, що під час навчання ми нормалізуємо *batch_size* прикладів, а під час тестування ми нормалізуємо статистику, отриману на основі всієї навчальної множини, тому що побачити заздалегідь тестові дані ми не можемо. А саме, ми обчислюємо маточікування та дисперсію для певного батча (пакету) $B = x_1, \dots, x_m$ наступним чином:

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i, \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

За допомогою цих статистичних характеристик ми перетворюємо функцію активації таким чином, щоб вона мала нульове маточікування та одиничну дисперсію на всьому батчі:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

де $\varepsilon > 0$ – параметр, що захищає нас від ділення на 0 (у разі, якщо середньоквадратичне відхилення батча дуже мало або навіть дорівнює нулю). Нарешті, щоб отримати остаточну функцію активації y , нам треба переконатися, що під час нормалізації ми не втратили здатності до узагальнення, і оскільки до вихідних даних ми застосували операції масштабування та зсуву, ми можемо дозволити довільне масштабування та зсув нормалізованих значень, отримавши остаточну функцію активації:

$$y_i = \gamma \hat{x}_i + \beta$$

де γ, β – це параметри батч-нормалізації, які система може навчити (їх можна оптимізувати шляхом градієнтного спуску на навчальних даних). Це узагальнення також означає, що батч-нормалізацію може бути корисно застосовувати безпосередньо до вхідних даних нейронної мережі.

Цей метод у застосуванні до глибоких згорткових мереж майже завжди успішно досягає своєї мети – прискорити навчання. Понад те, може стати відмінним регуляризатором, дозволяючи більш спокійно вибирати темп навчання, потужність L2-регуляризатора і dropout (іноді потреба у яких зовсім відпадає). Регуляризація тут – наслідок того факту, що результат роботи мережі для певного прикладу більше не детерміновано (він залежить від усього батча, в рамках якого цей результат отримано), що спрощує узагальнення.

І нарешті, хоча автори методу рекомендують застосовувати батч-нормалізацію до функції активації нейрона, останні дослідження показують, що якщо не корисніше, то, принаймні, так само вигідно використовувати її після активації.

Інші способи боротьби з перенавчанням

Рання зупинка (early stopping)

Поки що ми використовували валідаційну множину даних виключно для контролю за ходом навчання, що, без сумніву, не раціонально (оскільки ці дані не використовуються в конструктивних цілях). Насправді валідаційна множина може бути базою для оцінки гіперпараметрів мережі

(таких, як глибина, кількість нейронів/ядер, параметри регуляризації тощо). Уявімо, що мережа проганяється з різними комбінаціями гіперпараметрів, а потім рішення приймається на основі їхньої продуктивності на валідаційній множині. Звернімо увагу, що ми не повинні знати нічого про тестову множину до того, як остаточно визначимося з гіперпараметрами, оскільки інакше ознаки тестової множини мимоволі увіллються в процес навчання. Цей принцип також відомий як золоте правило машинного навчання і порушувалося в багатьох ранніх підходах.

Можливо, найпростіший спосіб використання валідаційної множини – налаштування кількості "епох" (циклів) за допомогою процедури, відомої як рання зупинка – просто зупиніть процес навчання, якщо за задану кількість епох (параметр *patience*) втрати не починають зменшуватися. Наприклад, якщо наш набір даних відносно невеликий і насичується швидко, ми встановимо *patience* рівним п'ять епох, а максимальну кількість епох збільшимо до 50 (навіть чи це число буде досягнуто).

Ансамблі (ensembles)

Одна цікава особливість нейронних мереж, яку можна помітити, коли вони використовуються для розподілу даних на більш ніж два класи — це те, що за різних початкових умов навчання їм легше дається розподіл за одними класами, тоді як інші помиляються. На прикладі відомого у задачах комп'ютерного навчання MNIST набору даних (рукописних чисел від 0 до 10) можна виявити, що окремо взята нейронна мережа чудово вміє відрізняти трійки від п'ятірок, але не вчиться правильно відокремлювати одиниці від сімок, тоді як із справи з іншою мережею йдуть навпаки.

З цією невідповідністю можна боротися за допомогою методу статистичних ансамблів – замість однієї мережі побудувати кілька її копій з різними початковими значеннями та обчислити їхній середній результат на тих самих вхідних даних.

Розширення навчальної множини (data augmentation)

У той час як описані вище методи стосувалися в основному тонкої настройки самої моделі, буває корисно вивчити варіанти налаштування даних, особливо коли йдеться про завдання розпізнавання зображень.

Припустимо, що ми навчили нейронну мережу розпізнавати рукописні цифри, які були приблизно одного розміру і були акуратно вирівняні. Тепер уявімо, що станеться, якщо хтось дасть цій мережі на тестування злегка зсунуті цифри різного розміру та нахилу – тоді її впевненість у правильному класі різко впаде. В ідеалі добре б вміти так навчати мережу, щоб вона залишалася стійкою до подібних перетворень, але наша модель може навчатися тільки на основі тих зразків, які ми їй надали, при тому, що вона проводить до певної міри статистичний аналіз навчальної множини і екстраполює його.

На щастя, для цієї проблеми існує рішення, просте, але ефективне, особливо на задачах розпізнавання зображень: штучно розширити навчальні дані версіями з штучними перетвореннями під час навчання! Це означає наступне: перед тим, як подати приклад на вхід моделі, ми застосуємо до нього всі трансформації, які вважаємо за потрібні, а потім дозволимо мережі безпосередньо спостерігати, який ефект має застосування їх до даних і навчаючи її “добре поводитись” і на цих приклади. Наприклад, кілька прикладів зрушених, масштабованих, деформованих, нахилених цифр з набору MNIST наведений на рисунку 4.11.

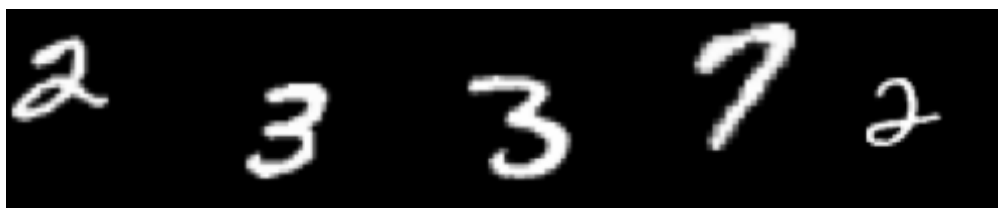


Рисунок 4.11 – Приклад деформованих чисел з набору даних MNIST

Ініціалізація мережі

Ініціалізація мережі – це принцип вибору початкових значень ваг для шарів, що становлять модель. Очевидно, що це питання дуже важливе: встановлення всіх ваг в 0 буде серйозною перешкодою для навчання,

оскільки жодна з ваг спочатку не буде активною. Надавати вагам значення з інтервалу ± 1 – теж зазвичай не найкращий варіант – насправді, іноді (залежно від завдання та складності моделі) від правильної ініціалізації моделі може залежати, чи досягне вона найвищої продуктивності або взагалі не сходитиметься. Навіть якщо завдання не передбачає такої крайності, вдало обраний спосіб ініціалізації ваги може значно впливати на здатність моделі до навчання, так як він встановлює параметри моделі з урахуванням функції втрат. Наведемо два найцікавіші методи.

Метод ініціалізації Зав'єра (Xavier) (іноді метод Glorot'a).

Основна ідея цього методу – спростити проходження сигналу через шар під час як прямого, так і зворотного поширення помилки для лінійної функції активації (цей метод також добре працює для сигмоїдної функції, оскільки ділянка, де вона ненасичена, також має лінійний характер). При обчисленні ваг цей метод спирається на імовірнісний розподіл (рівномірний або нормальний) з дисперсією, що дорівнює $var(W) = \frac{2}{n_{in} + n_{out}}$, де n_{in}, n_{out} – це кількість нейронів в попередньому і наступному шарах відповідно.

Метод ініціалізації Ге (He)

Це варіація методу Зав'єра, що більше підходить функції активації ReLU, що компенсує той факт, що ця функція повертає нуль для половини області визначення. А саме, у цьому випадку $Var(W) = \frac{2}{n_{in}}$.

Щоб отримати необхідну дисперсію для ініціалізації Зав'єра, розглянемо, що відбувається з дисперсією вихідних значень лінійного нейрона (без складової зсуву), припускаючи, що ваги та вхідні значення не корелюють і мають нульове маточікування:

$$\begin{aligned} Var\left(\sum_{i=1}^{n_{in}} w_i x_i\right) \\ = \sum_{i=1}^{n_{in}} Var(w_i x_i) = \sum_{i=1}^{n_{in}} Var(W) Var(X) = n_{in} Var(W) Var(X) \end{aligned}$$

З цього випливає, що щоб зберегти дисперсію вхідних даних після проходження через шар, необхідно, щоб дисперсія була $Var(W) = \frac{1}{n_{in}}$. Можемо застосувати цей аргумент при зворотному поширенні помилки, щоб отримати $Var(W) = \frac{1}{n_{out}}$. Так як зазвичай ми не можемо задовольнити обох цих вимог, ми вибираємо дисперсію ваги як їх середнє: $var(W) = \frac{2}{n_{in}+n_{out}}$, що на практиці, як правило, працює чудово.

Два цих методи підійдуть для більшості прикладів, з якими можна зіткнутися (хоча на дослідження також заслуговує на метод ортогональної ініціалізації (orthogonal initialization), особливо стосовно рекурентних мереж). Вказати спосіб ініціалізації для шару не складно: лише треба вказати параметр `init`, як описано нижче. Наприклад, можна використовувати рівномірну ініціалізацію Ге (`he_uniform`) для всіх шарів ReLU і рівномірну ініціалізацію Зав'єра (`glorot_uniform`) для вихідного softmax шару.

З практики

При використанні крос-валідації можна зробити такі висновки:

1. Якщо помилка велика на більшості ділянок, то, швидше за все, проблема в моделі.
2. Якщо дані навчальної вибірки характеризуються сильно зміщеними математичним очікуванням та дисперсією, то рівень узагальнення буде низьким, що може бути пов'язане з перенавчанням.
3. Якщо знайдені сильні відхилення на певних підвибірках, то, ймовірно, проблема на цих ділянках даних або модель недонавчена.
4. При малому обсязі навчальної вибірки крос-валідація може стати способом боротьби з перенавчанням.

ТЕМА №5: МЕТОД МАКСИМАЛЬНОЇ ПРАВДОПОДІБНОСТІ

Загальні визначення. Постановка завдання

У цьому розділі мова буде йти про те, що таке метод максимальної правдоподібності для оцінки параметрів на простому прикладі, щоб продемонструвати цей метод. Для розуміння матеріалу потрібні знання фундаментальних концепцій теорії ймовірності, таких як визначення спільної ймовірності та незалежності подій [19-22].

Часто в машинному навчанні ми використовуємо модель для опису процесу, результатом якого є дані, що спостерігаються. Наприклад, ми можемо використовувати модель випадкового лісу (random forest), щоб класифікувати, чи можуть клієнти скасувати підписку на послугу (відоме як моделювання відтоку), чи ми можемо використовувати лінійну модель для прогнозування доходу, який генеруватиметься для компанії залежно від того, скільки вони можуть витратити на рекламу (це буде приклад лінійної регресії). Кожна модель містить власний набір параметрів, які визначають, як модель виглядає.

Для лінійної моделі ми можемо записати це як $y = mx + c$. У цьому прикладі x може бути витратами на рекламу, а y може бути отриманим доходом. m і c – це параметри цієї моделі. Різні значення цих параметрів дадуть різні лінії (рисунок 5.1).

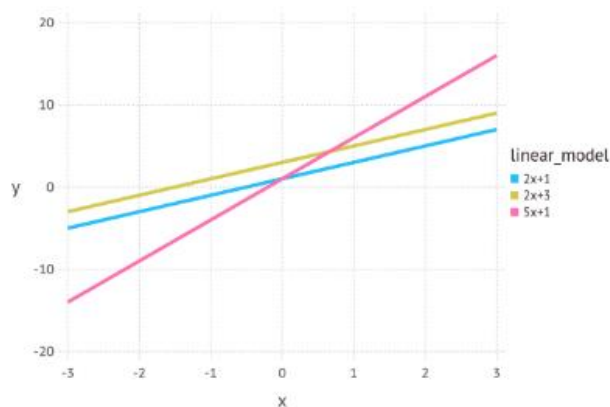


Рисунок 5.1 – Три лінійні моделі із різними значеннями параметрів.

Отже, параметри визначають план моделі. Тільки коли для параметрів вибрано конкретні значення, ми отримуємо екземпляр моделі, яка описує це явище.

Інтуїтивне пояснення оцінки максимальної правдоподібності

Оцінка максимальної правдоподібності – це метод, що визначає значення параметрів моделі. Значення параметрів підбираються так, щоб максимізувати ймовірність того, що процес, описаний моделлю, дав дані, які дійсно спостерігалися.

Наведене вище визначення може здатися трохи загадковим, тому розглянемо приклад, який допоможе зрозуміти це.

Припустимо, ми спостерігали 10 точок даних якогось процесу. Наприклад, кожна точка даних може представляти час у секундах, який потрібний студенту, щоб відповісти на конкретне запитання іспиту. Ці 10 точок даних показані на рисунку 5.2.

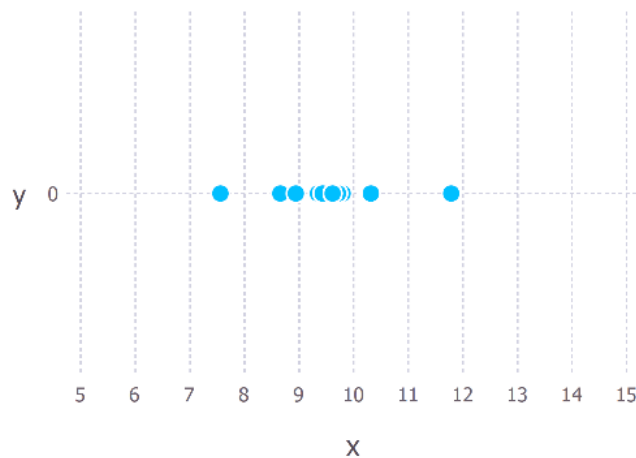


Рисунок 5.2 – 10 (гіпотетичних) точок даних, які спостерігалися

Спочатку маємо вирішити, яка модель, на нашу думку, найкраще описує процес генерації даних. Ця частина дуже важлива. Принаймні ми повинні мати гарне уявлення про те, яку модель використовувати. Зазвичай це відбувається через деякий досвід у предметній області, але ми не обговорюватимемо це тут.

Для цих даних ми припускаємо, що процес генерації даних може бути адекватно описаний гаусовим (нормальним) розподілом. Візуальний огляд рисунка 5.2 дозволяє припустити, що розподіл Гауса є правдоподібним, оскільки більшість із 10 точок згруповані в центрі, а кілька точок розкидані ліворуч і праворуч. (Не рекомендується приймати такого роду рішення на льоту, маючи всього 10 точок даних, але з огляду на те, що ми згенерували ці точки даних, ми будемо дотримуватися цього) [19-22].

Нагадаємо, що розподіл Гауса має 2 параметри. Середнє значення μ та стандартне відхилення σ . Різні значення цих параметрів призводять до різних кривих (як і прямі лінії на рисунку 5.1). Ми хочемо знати, яка крива, найімовірніше, була відповідальна за створення точок даних, які ми спостерігаємо (рисунк 5.3)? Оцінка максимальної правдоподібності – це метод, який знаходить значення μ та σ , які призводять до кривої, яка найкраще відповідає даним.

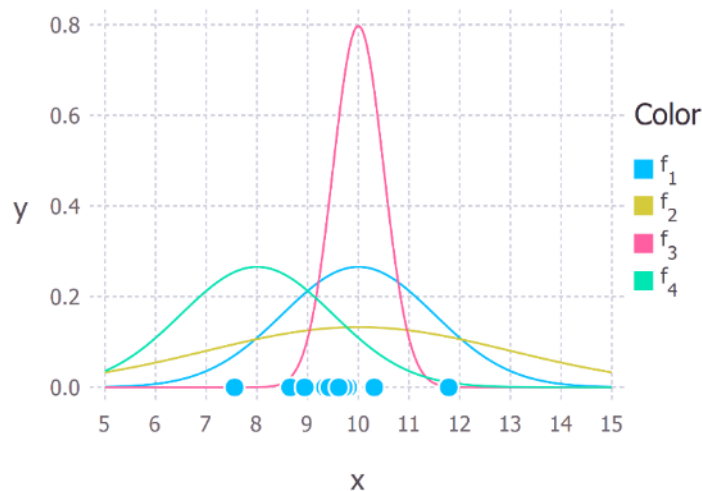


Рисунок 5.3 – 10 точок даних та можливі Гаусівські розподіли, з яких були взяті дані

f_1 нормально розподілений із середнім значенням 10 і дисперсією 2.25 (дисперсія дорівнює квадрату стандартного відхилення), це також позначається $f_1 \sim N(10, 2.25)$; $f_2 \sim N(10, 9)$; $f_3 \sim N(10, 0.25)$ і $f_4 \sim N(8, 2.25)$. Мета максимальної правдоподібності – знайти значення параметрів, що дають розподіл, що максимізує ймовірність спостереження даних.

Справжній розподіл, з якого були згенеровані дані, був $f_1 \sim N(10, 2.25)$, що є синьою кривою на рисунку 5.3.

Обчислення оцінок максимальної правдоподібності

Тепер, коли ми маємо інтуїтивне розуміння того, що таке оцінка максимальної правдоподібності, ми можемо перейти до навчання обчисленню значень параметрів. Знайдені значення називаються оцінками максимальної правдоподібності (MLE).

Продemonструємо це на простому прикладі. Припустимо, цього разу ми маємо три точки даних, і ми припускаємо, що вони були створені з процесу, який адекватно описується розподілом Гауса. Це точки 9, 9.5 та 11. Як ми обчислюємо оцінки максимальної правдоподібності значень параметрів Гаусівського розподілу μ та σ ? Ми хочемо розрахувати загальну ймовірність спостереження всіх даних, тобто спільний розподіл ймовірностей усіх точок даних, що спостерігаються. Для цього нам потрібно вирахувати деякі умовні ймовірності, що може виявитися дуже важким. Отже, тут ми зробимо наше перше припущення.

Передбачається, що кожна точка даних створюється незалежно від інших. Це значно спрощує математику. Якщо події (тобто процес, що генерує дані) незалежні, то загальна ймовірність спостереження всіх даних є продуктом спостереження кожної точки даних окремо (тобто добутком граничних ймовірностей).

Щільність ймовірності спостереження однієї точки даних x , що згенерована з розподілу Гауса, визначається як:

$$P(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)$$

Точка з комою, що використовується в позначенні $P(x; \mu, \sigma)$, призначена для того, щоб підкреслити, що символи, що з'являються після неї, є параметрами розподілу ймовірностей. Так що не слід плутати її з умовною ймовірністю (яка зазвичай представлена вертикальною лінією, наприклад, $P(A|B)$).

У нашому прикладі загальна (спільна) щільність ймовірності спостереження трьох точок даних визначається виразом:

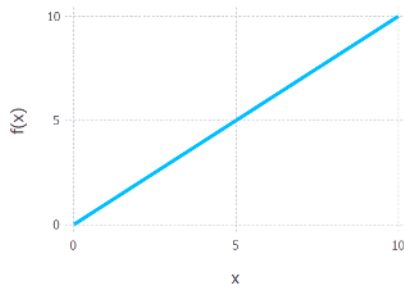
$$\begin{aligned} P(9, 9.5, 11; \mu, \sigma) \\ = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(9 - \mu)^2}{2\sigma^2}\right) \times \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(9.5 - \mu)^2}{2\sigma^2}\right) \\ \times \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(11 - \mu)^2}{2\sigma^2}\right) \end{aligned}$$

Нам просто потрібно з'ясувати значення μ та σ , які дають максимальне значення наведеного вище виразу.

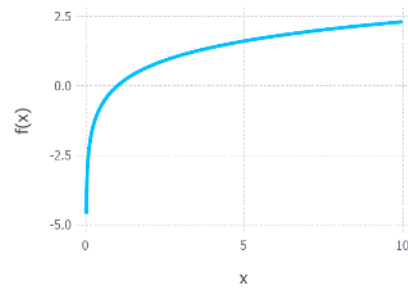
З вищої математики відомо, що є метод, який може допомогти нам знайти максимуми (і мінімуми) функцій. Це називається *диференціацією*. Все, що нам потрібно зробити, це знайти похідну функції, встановити похідну функцію рівною нулю, а потім змінити рівняння так, щоб параметр, що нас цікавить, став об'єктом рівняння. І ми матимемо значення MLE для наших параметрів.

The log likelihood

Вищезгаданий вираз для повної ймовірності насправді досить складно диференціювати, тому його майже завжди спрощують, беручи натуральний логарифм від цього виразу. Це абсолютно нормально, тому що натуральний логарифм є монотонно зростаючою функцією. Це означає, що якщо значення осі x збільшується, значення осі y також збільшується (рисунок 5.4). Це важливо, оскільки гарантує, що максимальне значення логарифму ймовірності зустрічається у тій самій точці, що й вихідна функція ймовірності. Отже, ми можемо працювати з більш простою логарифмічною правдоподібністю замість вихідної правдоподібності [19-22].



(a) $f(x) = x$



(b) $f(x) = \ln(x)$

Рисунок 5.4 – Монотонна поведінка вихідної функції, $y=x$ (зліва) та функції (натурального) логарифму $y=\ln(x)$ (праворуч). Обидві ці функції є монотонними, оскільки під час руху зліва направо по осі x значення y завжди збільшується.

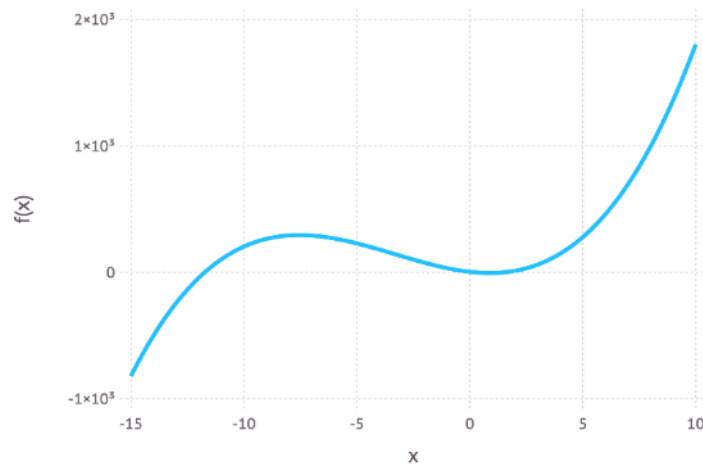


Рисунок 5.5 – Приклад немонотонної функції, тому що при русі зліва направо на графіку значення $f(x)$ збільшується, потім зменшується, а потім знову збільшується.

Логарифми вихідного виразу дають нам:

$$\begin{aligned} \ln P(x; \mu, \sigma) &= \ln \frac{1}{\sigma\sqrt{2\pi}} - \left(\frac{(9 - \mu)^2}{2\sigma^2} \right) + \ln \frac{1}{\sigma\sqrt{2\pi}} - \left(\frac{(9.5 - \mu)^2}{2\sigma^2} \right) \\ &+ \ln \frac{1}{\sigma\sqrt{2\pi}} - \left(\frac{(11 - \mu)^2}{2\sigma^2} \right) \end{aligned}$$

Цей вираз можна спростити, використовуючи закони логарифмів, щоб отримати:

$$\ln P(x; \mu, \sigma) = -3 \ln \sigma - \frac{3}{2} \ln 2\pi - \frac{1}{2\sigma^2} ((9 - \mu)^2 + (9.5 - \mu)^2 + (11 - \mu)^2)$$

Цей вираз можна диференціювати, щоб знайти максимум. У цьому прикладі ми знайдемо MLE середнього μ . Для цього візьмемо часткову похідну функції μ , що дає

$$\frac{\partial \ln P(x; \mu, \sigma)}{\partial \mu} = \frac{1}{\sigma^2} [9 + 9.5 + 11 - 3\mu]$$

Нарешті, встановлення лівої частини рівняння на нуль та подальше перетворення для μ дає:

$$\mu = \frac{9 + 9.5 + 11}{3} = 9.833$$

Це і є оцінка максимальної правдоподібності для μ . Ми можемо зробити те саме аналогічно з σ .

Корисні зауваження

Чи завжди можна точно вирішити оцінку максимальної правдоподібності?

Ні, це коротка відповідь. Більш ймовірно, що в реальному сценарії похідна від логарифмічної функції правдоподібності буде аналітично нерозв'язна (тобто надто складно/неможливо диференціювати функцію вручну). Тому ітераційні методи, такі як алгоритми очікування-максимізації використовуються для пошуку чисельних рішень для оцінок параметрів. Проте загальна ідея залишилася незмінною.

То чому максимальна ймовірність а не максимальна щільність?

Це просто педантичність статистики (але не дарма). Більшість людей схильні використовувати вірогідність та правдоподібність як взаємозамінні, але статистики та теоретики ймовірності проводять різницю між ними. Причину плутанини найкраще виявити, поглянувши на рівняння.

$$L(\mu, \sigma; data) = P(data; \mu, \sigma)$$

Ці вирази рівні! То що це означає? Давайте спочатку визначимо $P(\text{дані}; \mu, \sigma)$? Це означає "щільність ймовірності спостереження даних з

параметрами моделі μ і σ ". Варто зазначити, що ми можемо узагальнити це на будь-яку кількість параметрів та на будь-який розподіл.

З іншого боку, $L(\mu, \sigma; \text{дані})$ означає "ймовірність того, що параметри μ і σ приймуть певні значення, за умови, що ми спостерігали набір даних".

Наведене вище рівняння свідчить, що густина ймовірності даних з урахуванням параметрів дорівнює ймовірності параметрів з урахуванням даних. Але, незважаючи на те, що ці дві речі рівні, ймовірність та щільність ймовірності по суті ставлять різні питання: одне запитує про дані, а інше – про значення параметрів. Саме тому метод називається максимальною ймовірністю, а чи не максимальної щільністю.

Коли мінімізація методом найменших квадратів збігається з оцінкою максимальної правдоподібності?

Мінімізація методом найменших квадратів – ще один поширений метод оцінки значень параметрів моделі у машинному навчанні. Виявляється, коли модель вважається Гаусівською, як у наведених вище прикладах, оцінки MLE еквівалентні методу найменших квадратів.

Інтуїтивно ми можемо інтерпретувати зв'язок між двома методами, розуміючи їхні цілі. Для оцінки параметрів методом найменших квадратів хочемо знайти лінію, яка мінімізує загальний квадрат відстані між точками даних і лінією регресії (рисунок 5.6). Оцінюючи максимальну правдоподібність ми хочемо максимізувати загальну можливість даних. Коли передбачається розподіл гауса, максимальна ймовірність виявляється, коли точки даних наближаються до середнього значення. Оскільки розподіл Гауса є симетричним, це еквівалентно мінімізації відстані між точками даних та середнім значенням.

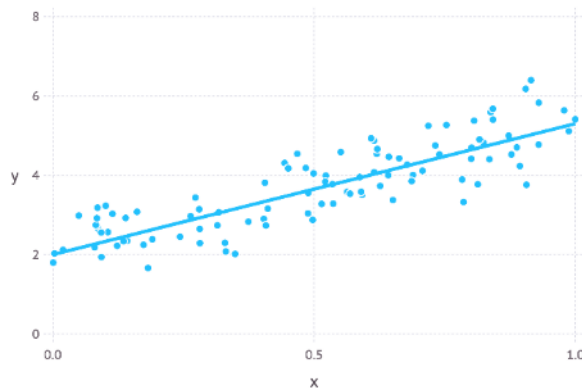


Рисунок 5.6 – Лінія регресії, що показує точки даних із випадковим гаусовим шумом

Звідки взялася двійкова функція втрати крос-ентронії?

У минулих розділах ми вже трохи розглядали це питання, та зпиралися на метод максимальної правдоподібності як на відомий факт. Давайте тепер, після вивчення самих засад методу, ще раз поглянемо на цю проблему.

Бінарна класифікація є унікальною проблемою, коли:

- 1.кожен приклад (x,y) належить до одного з двох додаткових класів
- 2.кожен приклад не залежить один від одного (тобто результат одного прикладу не впливає на результат іншого прикладу)

3.всі приклади генеруються з одного і того ж базового розподілу / процесу (тобто якщо ми створюємо набір даних для виявлення «кішка проти не-кішки», тоді всі приклади, які ми завантажуюємо в нейронну мережу для навчання «кішка проти не-кішки», повинні бути з того ж набору даних, а не з іншого незв'язаного набору даних, такого як «собака проти не-собаки»).

Нам потрібно лише прогнозувати для позитивного класу, тобто $P(y = 1|x) = \hat{p}$, тому що ймовірність для негативного класу може бути отримана з нього, тобто $P(y=0|x) = 1-P(y=1|x)=1-\hat{p}$:

$$P(y|x) = \begin{cases} \hat{p}, & \text{якщо } y = 1 \\ 1 - \hat{p}, & \text{якщо } y = 0 \end{cases}$$

Хороший двійковий класифікатор повинен давати високе значення $\hat{p}$, якщо в прикладі є позитивна мітка ($y=1$). З іншого боку, для прикладу з

негативною міткою ($y=0$) класифікатор повинен видавати низьке значення $\hat{p}$. Іншими словами:

- максимізувати $\hat{p}$, коли $y = 1$
- максимізувати $1 - \hat{p}$, коли $y = 0$.

Давайте подивимося, як ми можемо поєднати цю інтуїцію в один вираз:

$$P(y|x) = \hat{p}^y (1 - \hat{p})^{1-y}.$$

Виявляється, що такий однорядковий вираз, який ми навели, називається розподілом Бернуллі, а його обчислення для однієї точки даних називається випробуванням Бернуллі. Нам знову ж таки потрібно знайти критичні точки. Це досить просто, бо максимум (або мінімум) будь-якої опуклої функції (**u**-подібної функції) зустрічається у точках, де перша похідна дорівнює нулю, тобто (при min/max значеннях функції похідна дорівнює нулю)

$$\frac{\partial P(y|x)}{\partial p} = 0$$

Обчислення з виразом розподілу Бернуллі та її похідної може бути трохи складними у його поточної формі, не кажучи вже про те, що множення та зведення в ступінь малих значень може бути чисельно нестабільним і може викликати чисельне переповнення. На щастя, тут нам може допомогти натуральний логарифм (представлений "log", а не "ln").

Нагадаємо корисні властивості натурального логарифму:

$$\log(a * b) = \log a + \log b$$

$$\log(a^x) = x \log a$$

Тепер прологарифмуємо вираз Бернуллі:

$$\begin{aligned} \log P(y|x) &= \log \hat{p}^y (1 - \hat{p})^{1-y} = \log \hat{p}^y + \log(1 - \hat{p})^{1-y} \\ &= y \log \hat{p} + (1 - y) \log(1 - \hat{p}) \end{aligned}$$

Та подивимося на графік першого та другого члену цієї формули на рисунку 5.7

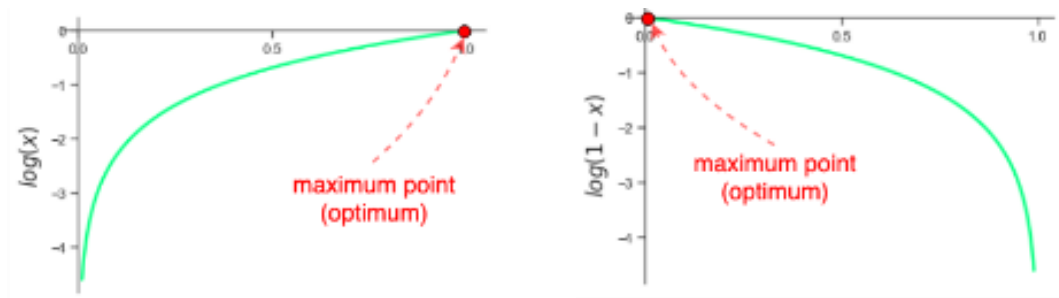


Рисунок 5.7 – Графік отриманого рівняння після логарифмування виразу Бернуллі: перший член – зліва, другий – праворуч

Зауважте, що після застосування натурального логарифму до розподілу Бернуллі ми спростили вираз до суми логарифму ймовірностей. Також зверніть увагу, що це спрощене вираження дуже схоже на функцію втрат двійкової крос-ентропії, але зі зворотними знаками. Щоб досягти максимальної точки логарифму розподілу Бернуллі чисельним методом (тобто ітеративним рухом у напрямку оптимальної точки), нам потрібно буде виконати «градієнтний підйом», коли крива логарифмічної функції згинається нагору, тобто увігнута. У нейронних мережах ми вважаємо за краще використовувати градієнтний спуск замість підйому, щоб знайти оптимальну точку. Ми робимо це, тому що навчання / оптимізація нейронних мереж ставиться як проблема «мінімізації втрат», тому саме тут ми додаємо негативний знак до прологарифмованого розподілу Бернуллі, результатом є функція двійкової крос-ентропійної втрати (BCE):

$$BCE(y, \hat{p}) = -\log P(y|x) = -y \log \hat{p} - (1 - y) \log(1 - \hat{p})$$

Нагадаємо графік функції двійкової крос-ентропії на рисунку 5.8.

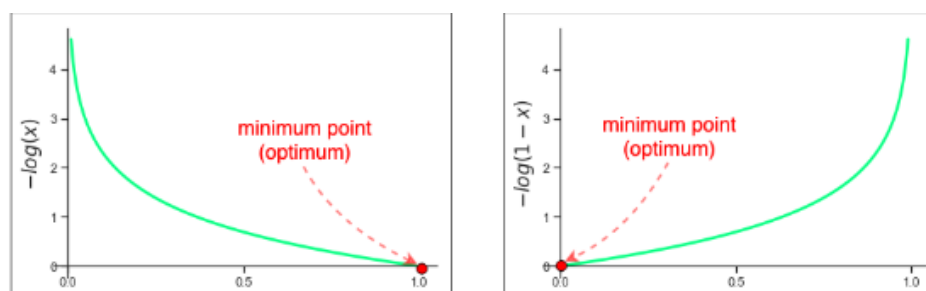


Рисунок 5.8 – Графік функції двійкової крос-ентропії. Зліва – перший член формули, праворуч – другий.

Точка мінімуму і точка максимуму знаходяться в одній точці, тепер ми можемо легко застосувати градієнтний спуск і рухатися вниз по кривій до оптимальної точки.

Нагадаємо також, що для однієї точки даних ми максимізуємо тест (випробування) Бернуллі (як ми зробили на прикладі вище), а для кількох точок даних – максимізуємо результат декількох тестів Бернуллі. Тобто нам потрібно буде взяти добуток ймовірностей, а все інше залишиться незмінним. Відповідно у кінцевій формулі лише з'явиться середня сума по всіх прикладах.

$$BCE(y, \hat{p}) = \frac{1}{m} \sum_{i=1}^m -y^{(i)} \log \hat{p}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{p}^{(i)})$$

На закінчення зазначимо, що наше припущення, що основні дані відповідають розподілу Бернуллі, дозволило нам використовувати MLE та придумати відповідну функцію витрат. Це припущення / знання даних називають "априорі" у байєсівській статистиці.

Оцінка максимальної ймовірності підкидання монети?

Розглянемо ще один дуже простий приклад оцінки максимальної правдоподібності ймовірностей помилкового підкидання монети, який також приведе нас до функції бінарної крос-ентропії, та може надати ще трохи інтуїції як працює метод.

Як ми вже знаємо, випробування Бернуллі – це випадковий експеримент із двома результатами. Стандартний приклад – це підкидання, можливо упередженої, монети. Позначимо спочатку результати за допомогою 0 та 1 замість орла та решки, оскільки це звучить набагато професійніше. Монета приземлиться на один бік, скажімо 1 з ймовірністю p . Вона приземлиться на іншій стороні 0 з ймовірністю $1 - p$, оскільки сума ймовірностей повинна дорівнювати одиниці. У математичному формулюванні отримуємо ймовірність $P(X=1) = p$ і $P(X=0) = 1 - p$. Тепер ми можемо визначити функцію маси ймовірності $f(x; p)$

$$f(x; p) = \begin{cases} p, & \text{якщо } x = 1 \\ 1 - p, & \text{якщо } x = 0 \end{cases}$$

Якщо нам відомі лише безперервні розподіли ймовірностей, функція маси ймовірності аналогічна функції густини ймовірності, але в дискретному випадку.

Тепер проведено оцінка максимальної правдоподібності. Припустимо, що ми кілька разів підкидали монету і отримали результат $x_1, \dots, x_n$ — які або 0 чи 1. Стохастичним мовою це називається спостереженнями. Питання в тому, яка ймовірність p . Інтуїтивно можна було припустити, що це кількість монет, що випали, розділена на загальну кількість кинутих монет. Наприклад, якщо ми кинули монету сто разів і 30 разів отримали 1, то, можливо, ми припустимо $p = 30/100$. Насправді це інтерпретація з частотної точки зору. Нижче ми доведемо, що інтуїція в цьому випадку є вірною, довівши, що припущення $p = \sum x_i / n$ є «найвірогіднішим» значенням реальної p .

По-перше, давайте переформулюємо нашу ймовірну функцію маси як

$$f(x; p) = p^x (1 - p)^{1-x},$$

що спрощує розрахунок за його допомогою. Якщо у нас є (незалежні) спостереження $x_1, \dots, x_n$, їх спільна функція мас ймовірності дорівнює

$$f(x_1, \dots, x_n; p) = \prod_i f(x_i; p) = p^{\sum x_i} (1 - p)^{n - \sum x_i}$$

Якщо інтерпретувати це як функцію не в спостереженнях з фіксованим параметром p , але натомість як функцію параметра моделі з фіксованими спостереженнями ми отримуємо те, що називається функцією правдоподібності $L(p; x_1, \dots, x_n) = f(x_1, \dots, x_n; p)$.

Тепер хочемо знайти p з найбільшою ймовірністю з урахуванням спостереження $x_1, \dots, x_n$, тобто хочемо максимізувати нашу функцію правдоподібності $L(p; x_1, \dots, x_n)$. У цьому випадку простіше максимізувати логарифм ймовірності (вище ми вже про це казали): $\log L(p) = \sum x_i \log p +$

$(n - \sum x_i) \log(1 - p)$, що дає той самий результат, тому що **log** монотонно зростає.

Як пам'ятаєте, максимізація функції означає встановлення її першої похідної на нуль.

$$\frac{\partial \log L(p)}{\partial p} = \frac{\sum x_i}{p} - \frac{n - \sum x_i}{1 - p} = \frac{(1 - p) \sum x_i - pn + p \sum x_i}{p(1 - p)} = \frac{\sum x_i - pn}{p(1 - p)} \equiv 0$$

Множення на $p(1-p)$ означає, що максимум досягається при $p = \sum x_i / n$. Це найбільш ймовірне значення для **p** враховуючи наше спостереження, яке підтверджує нашу інтуїцію.

ТЕМА №6: МЕТОДИ ОПТИМІЗАЦІЇ

Завдання знаходження оцінок параметрів динамічних моделей має досить довгу історію. Більшість робіт у цій галузі ґрунтується на ймовірнісному підході, коли оцінка невинпадкового параметра є випадковою величиною через прояви різних факторів. Існують методи для побудови точкових оцінок, наприклад метод максимальної правдоподібності. Як правило, при їх застосуванні досліджуються статистичні властивості цих оцінок: незміщення, збіжність до точного значення. У багатьох випадках недостатньо мати лише точкові оцінки, тому необхідно знати, у яких межах знаходиться точне значення параметра із заданою надійністю, що пов'язано з проблемою побудови довірчих інтервалів. Усі перелічені підходи засновані на тому, що є якась інформація про досліджуване явище, наприклад, відомі спостереження за реалізацією випадкової величини, тобто задана вибірка. За підсумками цієї інформації розроблялися алгоритми побудови оцінок. Іншим напрямом досліджень є застосування прямих методів оптимізації. Існує безліч класичних методів оптимізації нульового, першого та другого порядків, проте їх реалізація може бути пов'язана із завищеними вимогами до елементів постановки задачі. Альтернативою є застосування метаввристичних алгоритмів, які дозволяють отримати рішення досить хорошої якості за прийнятний час. У той самий час, зі зростанням кількості змінних рішення потребує значних обчислювальних ресурсів.

Найбільш відомі градієнтні методи оптимізації в машинному навчанні на даний момент наступні: метод стохастичного градієнтного спуску (Stochastic Gradient Descent, SGD), класичний метод моментів (Classical Momentum), прискорений градієнтний метод Нестерова (Nesterov Accelerated Gradient, NAG), Gradient, AdaGrad), метод ковзного середнього (Root Mean Square Propagation, RMSProp), метод адаптивної оцінки моментів (Adaptive Moment Estimation, Adam), модифікація методу Adam

(Adamax), прискорений за Нестеровим метод адаптивної оцінки моментів (Nesterov-accelerated Adaptive Moment Estimation, Nadam). Вони дозволяють знайти наближене рішення за прийнятний час під час роботи з великими даними. Як правило, методи мають багато параметрів, які потрібно вибрати, і для різних завдань ці параметри можуть відрізнятися [26-34].

Гرادієнтний спуск.

Постановка задачі

Перш ніж описувати метод, слід спочатку описати завдання, а саме: «Дані безліч K і функція $f: K \rightarrow \mathbb{R}$, потрібно знайти точку $x^* \in K$, таку, що $f(x) \geq f(x^*)$ для всіх $x \in K$ », що зазвичай записується, наприклад, ось так

$$f(x) \rightarrow \min_{x \in K}.$$

Теоретично зазвичай передбачається, що f – диференційована і опукла функція, а K – опукла безліч (а ще краще, якщо взагалі $K = \mathbb{R}^n$), це дозволяє дати якісь гарантії успішності застосування градієнтного спуску. Насправді градієнтний спуск успішно застосовується навіть коли завдання немає жодного з перелічених вище властивостей.

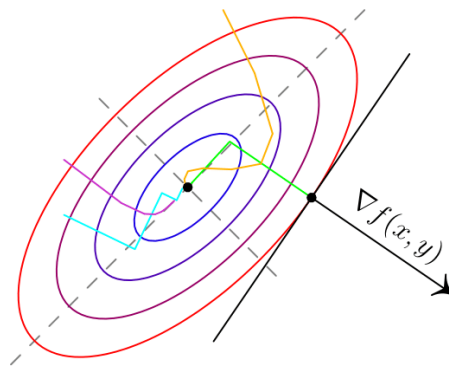


Рисунок 6.1 – Схематичне визначення градієнта

Допустимо поки що нам потрібно просто знайти мінімум одновимірної функції

$$f(x) \rightarrow \min_{x \in \mathbb{R}}$$

Ще в 17 столітті П'єром Ферма був придуманий критерій, який дозволяв вирішувати прості завдання оптимізації, а саме, якщо x^* – точка мінімуму f^* , то

$$f'(x^*)=0$$

де f' – це похідна f . Цей критерій ґрунтується на лінійному наближенні

$$f(x) \approx f(x^*) + f'(x^*)(x - x^*).$$

Чим ближче x до x^* , то точніше це наближення. У правій частині – вираз, який при $f'(x^*) \neq 0$ може бути як більшим $f(x^*)$ так і меншим – це основна суть критерію. У багатовимірному випадку аналогічно з лінійного наближення $f(x) \approx f(x^*) + \nabla f(x^*)^T (x - x^*)$ (тут і далі $x^T y = \sum_{i=1}^n x_i y_i$ – це стандартний скалярний добуток, форма запису обумовлена тим, що скалярний добуток – це те саме, що і матричний добуток вектор-рядку на вектор-стовпець) виходить критерій

$$\nabla f(x^*) = 0.$$

Розмір $\nabla f(x^*)$ – це градієнт функції f у точці x^* . Також рівність градієнта нулю означає рівність всіх часткових похідних нулю, тому в багатовимірному випадку можна отримати цей критерій просто послідовно застосувавши одновимірний критерій до кожної змінної окремо.

Варто зазначити, що зазначені умови є необхідними, але не достатніми, найпростіший приклад – 0 для функцій $f(x) = x^2$ і $f(x) = x^3$ (рисунок 6.2).

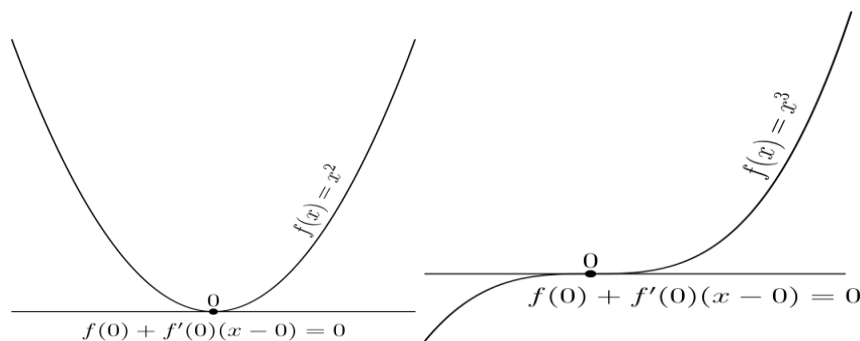


Рисунок 6.2 – Приклад градієнта функцій $f(x) = x^2$ і $f(x) = x^3$

Цей критерій є достатнім у разі опуклої функції, багато в чому через це для опуклих функцій вдалося отримати так багато результатів.

Квадратичні функції

Квадратичні функції у просторі R^n – це функція виду

$$f(x) = f(x_1, x_2, \dots, x_n) = \frac{1}{2} \sum_{i,j=1}^n a_{ij} x_i x_j - \sum_{i=1}^n b_i x_i + c$$

Для економії місця (та й щоб менше возитися з індексами) таку функцію зазвичай записують у матричній формі:

$$f(x) = \frac{1}{2} x^T A x - b^T x + c$$

де $x = (x_1, \dots, x_n)^T$, $b = (b_1, \dots, b_n)^T$, A – матриця, у якої на перетині i -го рядка і j -го стовпця стоїть величина $0.5(a_{ij} + a_{ji})$ (матриця A при цьому виходить симетричною – це важливо!). Далі при згадці квадратичної функції ми матимемо зазначену вище функцію.

Навіщо це? Справа в тому, що квадратичні функції важливі в оптимізації з двох причин:

1. Вони теж зустрічаються на практиці, наприклад, при побудові лінійної регресії методом найменших квадратів.
2. Градієнт квадратичної функції – це лінійна функція, зокрема для функції вище маємо

$$\frac{\partial}{\partial x_i} f(x_1, x_2, \dots, x_n) = a_{ij} x_j + \sum_{j \neq i} \frac{1}{2} (a_{ij} + a_{ji}) x_j - b_i$$

або в матричній формі

$$\nabla f(x) = A x - b$$

Таким чином, система $\nabla f(x) = 0$ – лінійна система. Системи, простіше за лінійну, просто не існує. Думка, до якої ми намагалися підібратися – оптимізація квадратичної функції – найпростіший клас завдань оптимізації. З іншого боку, те що, що $\nabla f(x^*) = 0$ – необхідні умови мінімуму дає можливість вирішувати лінійні системи через завдання оптимізації.

Корисні властивості градієнта

Добре, ми начебто з'ясували, що якщо функція диференційована (у неї існують похідні по всіх змінних), то в точці мінімуму градієнт повинен дорівнювати нулю. А ось чи несе градієнт якусь корисну інформацію у разі, коли він відмінний від нуля?

Спробуємо поки вирішити простішу задачу: дана точка x , знайти точку $\bar{x}$ таку, що $f(\bar{x}) < f(x)$. Давайте візьмемо точку поряд з x , знову ж таки використовуючи лінійне наближення

$$f(\bar{x}) \approx f(x) + \nabla f(x)^T (\bar{x} - x)$$

Якщо взяти $\bar{x} = x - \alpha \nabla f(x)$, $\alpha > 0$ то ми отримаємо

$$f(\bar{x}) \approx f(x) + \alpha \|\nabla f(x)\|^2 < f(x)$$

Аналогічно, якщо $\alpha < 0$, то $f(x^-)$ буде більше $f(x)$ тут і далі

$$\|x\| = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$$

Знову ж таки, оскільки ми використовували наближення, ці міркування будуть вірні лише малих α . Підсумовуючи сказане вище, якщо $\nabla f(x) \neq 0$, то **градієнт вказує напрямок найбільшого локального збільшення функції [26-28]**.

Два приклади для двовимірних функцій наведено на рисунку 6.3. Такі картинки можна часто побачити в демонстраціях градієнтного спуску. Кольорові лінії – це так звані **лінії рівня**, це безліч точок, для яких функція набуває фіксованих значень, в нашому випадку це кола та еліпси. Синім позначені лінії рівня з нижчим значенням, червоними – з вищим.

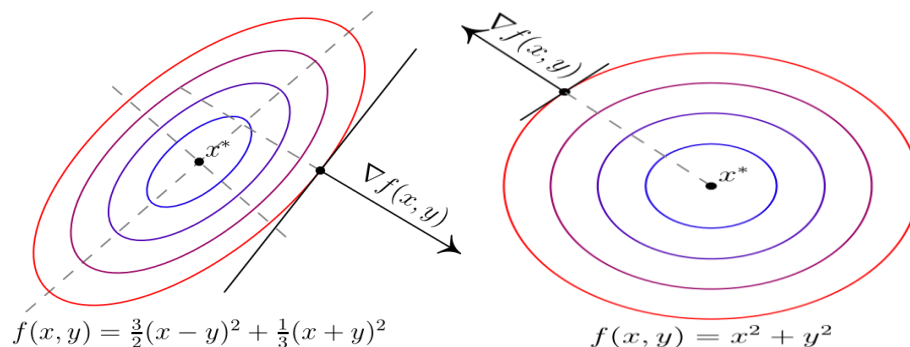


Рисунок 6.3 – Приклади двовимірних функцій та градієнтів

Зверніть увагу, що для поверхні, заданої рівнянням виду $f(x)=c$, $\nabla f(x)$ задає нормаль (перпендикуляр) до цієї поверхні. Також зверніть увагу, що хоча градієнт і показує у напрямку найбільшого збільшення функції, немає жодної гарантії, що у напрямку, зворотному до градієнта, можна знайти мінімум (приклад – ліва картинка на рисунку 6.3).

Градiєнтний спуск (ГС)

До базового методу градієнтного спуску залишився лише малий крок: ми навчилися по точці x отримувати точку x з меншим значенням функції f . Що заважає нам повторити це кілька разів? По суті це і є градієнтний спуск: будуємо послідовність

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k)$$

Розмір α_k називається **розміром кроку** (у машинному навчанні – швидкість навчання). Пару слів щодо вибору α_k : якщо α_k – дуже маленькі, то послідовність повільно змінюється, що робить алгоритм не дуже ефективним; якщо ж α_k дуже великі, то лінійне наближення стає поганим, а може, навіть і невірним. На практиці розмір кроку часто підбирають емпірично, теоретично зазвичай передбачається липшицевість градієнта, а саме, якщо

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|$$

для всіх x, y , то $\alpha_k < 2/L$ гарантує спадання $f(x_k)$.

Аналіз для квадратичних функцій

Якщо A – симетрична оборотна матриця, $Ax^* = b$, то для квадратичної функції

$$f(x) = \frac{1}{2} x^T A x - b^T x + c$$

точка x^* є точкою мінімуму (за умови, що цей мінімум взагалі існує – f не набуває скільки завгодно близьких до $-\infty$ значення тільки якщо A позитивно визначено), а для методу градієнтного спуску можна отримати таке

$$\begin{aligned} x_{k+1} - x^* &= x_k - \alpha_k \nabla f(x_k) - x^* = x_k - \alpha_k (Ax_k - b) - x^* \\ &= (x_k - x^*) - \alpha_k A(x_k - x^*) = (I - \alpha_k A)(x_k - x^*) \end{aligned}$$

де I – одинична матриця, тобто $Ix=x$ для всіх x . Якщо ж $\alpha_k \equiv \alpha$, то вийде

$$\|x_k - x^*\| = \|(I - \alpha A)^k(x_0 - x^*)\| \leq \|I - \alpha A\|^k \|x_0 - x^*\|$$

Вираз ліворуч – це відстань від наближення, отриманого на кроці k градієнтного спуску до точки мінімуму, праворуч – вираз виду $\lambda^k \beta$, що сходиться до нуля, якщо $|\lambda| < 1$ (умова, яку ми писали на α у попередньому пункті якраз це гарантує). Ця базова оцінка гарантує, що градієнтний спуск сходиться.

Градієнтний спуск з практичної точки зору

Градієнтний спуск – це метод знаходження мінімального значення функції втрат (як ми вже знаємо, існує безліч видів цієї функції). Мінімізація будь-якої функції означає пошук найглибшої западини у цій функції. Майте на увазі, що функція використовується для контролю помилки в прогнозах моделі машинного навчання. Пошук мінімуму означає отримання найменшої можливої помилки або підвищення точності моделі. Ми збільшуємо точність, перебираючи набір навчальних даних при налаштуванні параметрів нашої моделі (ваги та зміщення) [26-29].

Отже, градієнтний спуск потрібен мінімізації функції втрат.

Суть алгоритму – процес отримання найменшого значення помилки. Аналогічно це можна розглядати як спуск у западину у спробі знайти золото на дні ущелини (найнижче значення помилки, рисунок 6.4).

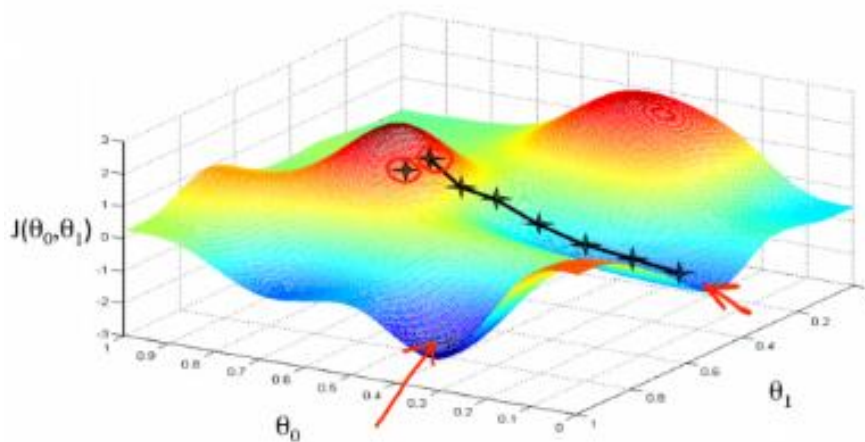


Рисунок 6.4 – Приклад знаходження мінімуму функції

Пошук мінімуму функції

Надалі, щоб знайти найнижчу помилку (найглибшу западину) у функції втрат (стосовно однієї ваги), потрібно налаштувати параметри моделі. Як ми їх налаштовуємо? У цьому допоможе математичний аналіз. Завдяки аналізу ми знаємо, що нахил графіка функції – похідна від функції змінної. Це нахил завжди вказує на найближчу впадину!

На рисунку 6.5 ми бачимо графік функції втрат (названий "Помилка" із символом "J") з однією вагою. Тепер, якщо ми обчислимо нахил (позначимо це як dJ/dw) функції втрат щодо однієї ваги, то отримаємо напрямок, у якому потрібно рухатися, щоб досягти локальних мінімумів. Давайте поки що уявимо, що наша модель має тільки одну вагу.

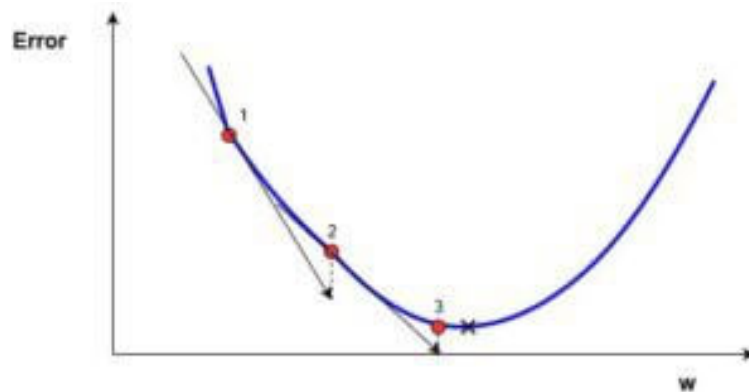


Рисунок 6.5 – Функція втрат та помилка

Функція втрат

Важливо: коли ми перебираємо всі навчальні дані, продовжуємо додавати значення dJ/dw для кожної ваги. Оскільки втрати залежать від прикладу навчання, dJ/dw також продовжують змінюватися. Потім ділимо зібрані значення на кількість прикладів навчання щоб одержати середнє значення. Потім ми використовуємо це середнє значення (для кожної ваги) для налаштування кожної ваги.

Також звернімо увагу: Функція втрат призначена для відстеження помилки з кожним прикладом навчання, тоді як похідна функції відносно

однієї ваги – це те, куди потрібно змістити вагу, щоб мінімізувати її для цього прикладу навчання. Ви можете створювати моделі навіть без використання функції втрат. Але вам доведеться використовувати похідну щодо кожної ваги (dJ/dw) [4-5].

Тепер, коли ми визначили напрямок, у якому потрібно підштовхнути вагу, нам треба зрозуміти, як це зробити. Тут ми використовуємо коефіцієнт швидкості навчання, його називають **гіперпараметром**. Гіперпараметр – це значення, необхідне вашою моделлю, про яке ми дійсно маємо дуже невиразне уявлення. Зазвичай ці значення можуть бути вивчені методом спроб та помилок. Тут не так: одне підходить для всіх гіперпараметрів. Коефіцієнт швидкості навчання можна розглядати як «крок у правильному напрямі», де напрям походить від dJ/dw .

Це була функція втрат, побудована на одну вагу (рисунок 6.5). У реальній моделі ми робимо все вище перераховане для всіх ваг, перебираючи всі приклади навчання. Навіть у відносно невеликій моделі машинного навчання у вас буде більш ніж 1 або 2 ваги. Це ускладнює візуалізацію, оскільки графік матиме розміри, які розум не може собі уявити.

Детальніше про градієнти

Крім функції втрат градієнтний спуск також вимагає градієнт, що є dJ/dw (похідна функції втрат щодо однієї ваги, виконана по всім вагам). dJ/dw залежить від вибору функції втрат. Найбільш поширена функція втрат задачі регресії - це середньоквадратична помилка, розглянемо її приклад

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

Похідна цієї функції щодо будь-якої ваги (ця формула показує обчислення градієнта для лінійної регресії):

$$\frac{\partial}{\partial \theta_j} MSE(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T x^{(i)} - y^{(i)}) x_j^{(i)}$$

Це – вся математика у ГС. Дивлячись на це можна сказати, що по суті ГС не містить багато математики. Єдина математика, яку він містить у собі – множення та розподіл. Це означає, що вибір функції вплине на обчислення градієнта кожної ваги.

Коефіцієнт швидкості навчання

Проблема у більшості моделей виникає з коефіцієнтом швидкості навчання. Погляньмо на оновлений вираз для кожної ваги (j лежить в діапазоні від 0 до кількості ваг, а θ_j це j -та вага у векторі ваг, k лежить в діапазоні від 0 до кількості зсувів, де B_k – це k -е зміщення у векторі зсувів). Тут α – це коефіцієнт швидкості навчання. З цього можна сказати, що ми обчислюємо $\frac{\partial J}{\partial \theta_j}$ (градієнт ваги θ_j), а потім крок розміру α в цьому напрямку. Отже, ми спускаємося градієнтом. Щоб оновити член зміщення треба тільки оновити θ_j на B_k .

$$\text{Робимо доки не досягнемо сходимості: } \theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Якщо обраний розмір кроку α занадто великий, ми подолаємо мінімум: тобто промахнемося повз мінімум. Якщо α занадто мала, ми використовуємо занадто багато ітерацій, щоб дістатися до мінімуму. Отже, α має бути оптимальною (рисунок 6.6).

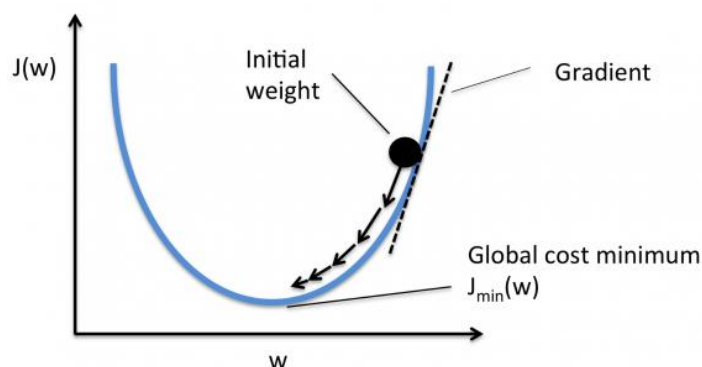


Рисунок 6.6 – Приклад градієнтного спуску на основі MSE функції втрат

Різні типи градієнтного спуску

Існує 3 варіанти градієнтного спуску (в минулому розділі ми трохи описували їх, рисунок 6.7):

1. **Mini-batch**: тут замість перебирання всіх прикладів навчання і з кожною ітерацією, що виконує обчислення тільки на одному прикладі навчання, ми обробляємо n навчальних прикладів відразу. Цей вибір хороший для великих наборів даних.

2. **Стохастичний градієнтний спуск**: у цьому випадку замість використання та зациклювання на кожному прикладі навчання, ми використовуємо один раз один приклад. Є кілька речей зауважень:

- З кожною ітерацією ГС вам потрібно перемішати набір навчання та обрати один випадковий приклад навчання.
- Оскільки ви використовуєте лише один приклад навчання, ваш шлях до локальних мінімумів буде дуже нестабільним, як у «п'яної людини», яка багато випила.

3. **Batch ГС**: Цикл на кожному прикладі навчання.

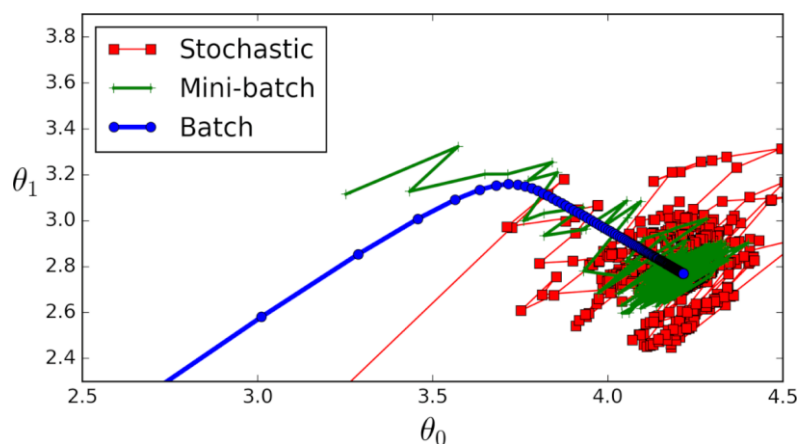


Рисунок 6.7 – Приклад навчання різними типами градієнтного спуску

Стохастичний градієнтний спуск (SGD або ГС)

Єдина формальна відмінність від звичайного градієнтного спуску градієнта – використання замість градієнта, функції $g(x, \theta)$ такої, що

$$E_{\theta}g(x, \theta) = \nabla f(x)$$

де E_{θ} – це математичне очікування за випадковою величиною θ), таким чином стохастичний градієнтний спуск має вигляд

$$x_{k+1} = x_k - \alpha_k g(x_k, \theta_k)$$

де θ_k – це деякий випадковий параметр, на який ми не впливаємо, але при цьому ми в середньому йдемо проти градієнта. Як приклад розглянемо функції

$$f(x) = \frac{1}{2m} \sum_{j=1}^m \|x - y_j\|^2, \nabla f(x) = \frac{1}{m} \sum_{i=1}^m (x - y_i); g(x, i) = x - y_i$$

Якщо i набуває значення $1, \dots, m$ рівномірно, то саме в середньому g – це градієнт f . Цей приклад показовий ще й наступним: складність обчислення градієнта в m разів більша, ніж складність обчислення g . Це дозволяє стохастичному градієнтному спуску робити за один і той же час у m разів більше ітерацій. Незважаючи на те, що стохастичний градієнтний спуск зазвичай сходиться повільніше за звичайний, за рахунок такого великого збільшення числа ітерацій виходить поліпшити швидкість збіжності на одиницю часу. На даний момент стохастичний градієнтний спуск є базовим методом навчання більшості нейронних мереж, реалізований у всіх основних бібліотеках з ML: tensorflow, torch, caffe, CNTK і т.д.

Варто відзначити, що ідеї *інерційних* методів застосовуються для стохастичного градієнтного спуску і на практиці часто дають приріст, теоретично зазвичай вважається, що асимптотична швидкість збіжності не змінюється через те, що основна похибка в стохастичному градієнтному спуску обумовлена дисперсією g .

Модифікації градієнтного спуску

Тепер проаналізуємо модифікації градієнтного спуску, що часто використовуються [26-33].

Інерційні чи прискорені градієнтні методи

Одна із проблем з ГС (чи СГС) полягає у тому, що коли функція потрапляє у “яр”, тобто по одному з напрямків маємо швидкий спуск, а по іншому повільний, то ГС призводить до осциляції і вкрай повільної збіжності до мінімуму.

Для вирішення цієї проблеми було запропоновано підхід, який збільшує крок у напрямку до мінімуму, і зменшує осциляцію. Це досягається за рахунок того, що зсув параметрів розраховується як виважена сума зсуву на попередньому кроці та нового на основі градієнта (рисунок 6.8).

Швидкість руху в напрямку мінімуму збільшується (бо цей напрямок присутній у всіх градієнтах), а осциляція гаситься. Ваговий параметр зазвичай вибирається рівним 0.9 або близько до того.

Загалом, усі методи такого класу виражаються у такому вигляді

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k) + \beta_k (x_k - x_{k-1})$$

Останній доданок характеризує цю саму «інерційність», тобто алгоритм на кожному кроці намагається рухатися проти градієнта, але при цьому за інерцією частково рухається в тому ж напрямку, що й на попередній ітерації. Такі методи мають дві важливі властивості:

1. Вони мало ускладнюють простий градієнтний спуск в обчислювальному плані.
2. При акуратному підборі α_k , β_k такі методи набагато швидші, ніж звичайний градієнтний спуск навіть з оптимально підібраним кроком.

Один з перших таких методів з'явився в середині 20-го століття і називався **метод важкої кульки**, що передавало природу інерційності методу: у цьому методі α_k , β_k не залежать від k і акуратно підбираються залежно від цільової функції. Варто зазначити, що α_k може бути будь-якою, а β_k – зазвичай лише трохи менше одиниці.

Метод важкої кульки – це найпростіший інерційний метод, але не найперший.

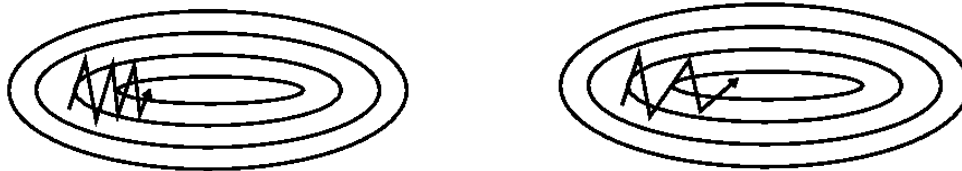


Рисунок 6.8 – Схематично робота СГС без імпульсу (зліва) та з імпульсом (праворуч)

При СГС з імпульсом (або SGD з імпульсом) використовується ковзаючий середній градієнт на кожній новій ітерації оптимізації. Рух у напрямку середніх минулих градієнтів додає імпульсного ефекту до алгоритму оптимізації, що дозволяє регулювати напрямки наступного кроку щодо історично домінуючого напрямку. Для цих цілей достатньо використовувати приблизний ковзаючий середній показник (sliding average), а не зберігати всі попередні значення градієнтів, щоб обчислити "чесне середнє значення". Формула, що налаштовує цей підхід наступна:

$$v(t) = \alpha v(t-1) + \eta \nabla_{\theta} J(\theta(t)), \theta(t+1) = \theta(t) - v(t)$$

де $v(t)$ – це накопичені середні градієнти на кроці t , коефіцієнт α знаходиться в діапазоні $[0,1]$ – потрібен для зберігання історії значень середнього (зазвичай вибирається близьким до 0.9).

Метод Чебишева

Перший метод такого типу був винайдений ще Чебишевим для вирішення систем лінійних рівнянь. У якийсь момент під час аналізу градієнтного спуску було отримано таку рівність

$$\begin{aligned} x_{k+1} - x^* &= (I - \alpha_k A)(x_k - x^*) = \dots \\ &= (I - \alpha_k A)(I - \alpha_{k-1} A) \dots (I - \alpha_1 A)(x_0 - x^*) = P_k(A)(x_0 - x^*) \end{aligned}$$

де P_k – це деякий багаточлен ступеня k . Чому б не спробувати підібрати α_k таким чином, щоб $P_k(A)(x_0 - x^*)$ було менше? Один з універсальних багаточленів, які найменше відхиляються від нуля – багаточлен Чебишева. Метод Чебишева насправді у тому, щоб підібрати параметри спуску так,

щоб P_k був многочленом Чебишева. Є, правда, одна невелика проблема: для звичайного градієнтного спуску це просто неможливо. Однак для інерційних методів це виявляється можливим. В основному це відбувається через те, що багаточлени Чебишева задовольняють рекурентному співвідношенню другого порядку

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

тому їх неможливо побудувати для градієнтного спуску, який обчислює нове значення лише по одному попередньому, а для інерційних стає можливим за рахунок того, що використовується два попередні значення. При цьому виявляється, що складність обчислення α_k, β_k не залежить ні від k , ні від розміру простору n .

Метод сполучених градієнтів

Ще один дуже цікавий і важливий факт (наслідок теореми Гамільтона-Келі): для будь-якої квадратної матриці A розміру $n \times n$ існує багаточлен P ступеня не більше n для якого $P(A)=0$. Чим це цікаво? Вся справа в тій самій рівності

$$x_{k+1} - x^* = P_k(A)(x_0 - x^*)$$

Якби ми могли підбирати розмір кроку в градієнтному спуску так, щоб отримувати саме цей багаточлен, що обнулює, то градієнтний спуск сходився б за фіксоване число ітерацій не більше розмірності A . Як ми уже з'ясували, для градієнтного спуску ми так робити не можемо. На щастя, для інерційних методів можемо. Опис та обґрунтування методу досить технічне, обмежимося суттю: на кожній ітерації вибираються параметри, що дають найкращий багаточлен, який можна побудувати з огляду на всі зроблені до поточного кроку вимірювання градієнта. При цьому

1. Одна ітерація градієнтного спуску (без урахування обчислень параметрів) здійснить одне множення матриці на вектор і 2-3 додавання векторів;

2. Обчислення параметрів також вимагає 1-2 множення матриці на вектор, 2-3 скалярних множення вектор на вектор і кілька додавань векторів.

Найскладніше? в обчислювальному плані – це множення матриці на вектор, зазвичай це робиться за час $O(n^2)$, проте зі спеціальною реалізацією це можна зробити за $O(m)$, де m – це кількість ненульових елементів в A . Враховуючи збіжність методу сполучених градієнтів не більше, ніж за n ітерацій, отримуємо загальну складність алгоритму $O(nm)$, що у всіх випадках не гірше $O(n^3)$ для методу Гауса або Холеського, але набагато краще у випадку, якщо $m \ll n^2$, що не так рідко зустрічається.

Метод сполучених градієнтів добре працює і у випадку, якщо f не є квадратичною функцією, але при цьому вже не сходиться за кінцеве число кроків і часто вимагає невеликих додаткових модифікацій.

Метод Нестерова

Наступне поліпшення методу ГС, що збільшує швидкість збіжності, було запропоновано в роботі Нестерова. Замість того, щоб вираховувати градієнт у поточній точці, використовуватимемо градієнт у точці “передбаченої” на підставі зсуву, розрахованого на попередньому кроці.

Для спільнот математичної оптимізації та машинного навчання прізвище «Нестеров» вже давно стало загальним. У 80-х роках минулого століття Ю.Є. Нестеров вигадав цікавий варіант інерційного методу, який має вигляд

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k + \beta_k(x_k - x_{k-1})) + \beta_k(x_k - x_{k-1})$$

або

$$v(t) = \alpha v(t-1) + \eta \nabla_{\theta} J(\theta(t) - \alpha v(t-1)), \theta(t+1) = \theta(t) - v(t)$$

при цьому не передбачається якогось складного підрахунку α_k, β_k як у методі сполучених градієнтів. Загалом поведінка методу схожа на метод важкої кульки, але при цьому його збіжність зазвичай набагато надійніша як у теорії, так і на практиці.

На рисунку 6.9 та 6.10 показані відмінності метода СГС з моментом та модифікації Нестерова

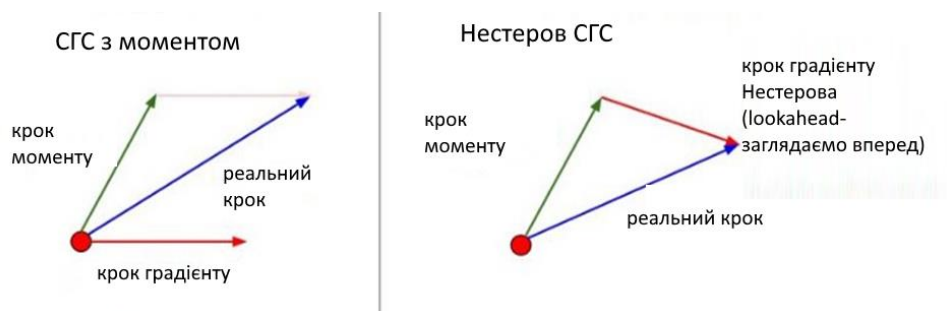


Рисунок 6.9 – Відмінності метода СГС з моментом та модифікації Нестерова

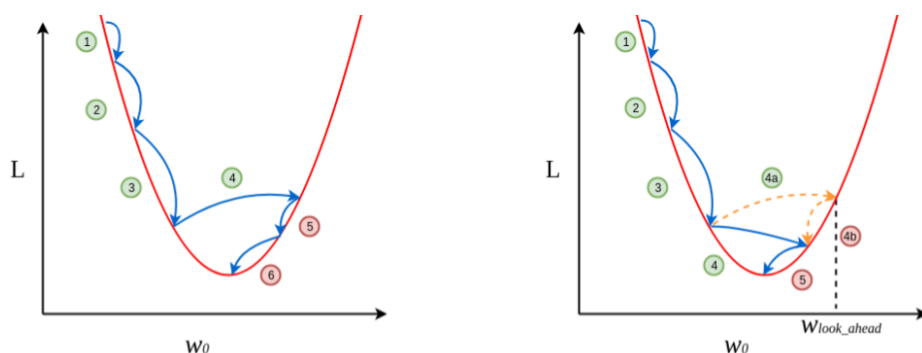


Рисунок 6.10 – Відмінності метода СГС з моментом (зліва) та модифікації Нестерова (праворуч) на схематичному прикладі

Субградієнтний спуск

Ця варіація дозволяє працювати з недиференційованими функціями. Знову згадаємо лінійне наближення – справа в тому, що є проста характеристика опуклості через градієнт, функція f , що диференціюється, опукла тоді і тільки тоді, коли виконується

$$f(y) \geq f(x) + \nabla f(x)^T (y - x)$$

для всіх x, y . Виявляється, що опукла функція не повинна бути диференційованою, але для будь-якої точки x обов'язково знайдеться такий вектор g , що

$$f(y) \geq f(x) + g^T (y - x)$$

для всіх y . Такий вектор g прийнято називати субградієнтом f у точці x . Множину всіх субградієнтів у точки x називають субдиференціалом x і позначають $\partial f(x)$ (попри позначення – немає нічого спільного з частковими

похідними). В одновимірному випадку g – це число, а вищевказана властивість просто означає, що графік f лежить вище за пряму, яка проходить через $(x, f(x))$ і має тангенс кута нахилу g (рисунок 6.11). Зазначимо, що субградієнтів однієї точки може бути кілька, навіть нескінченне число.

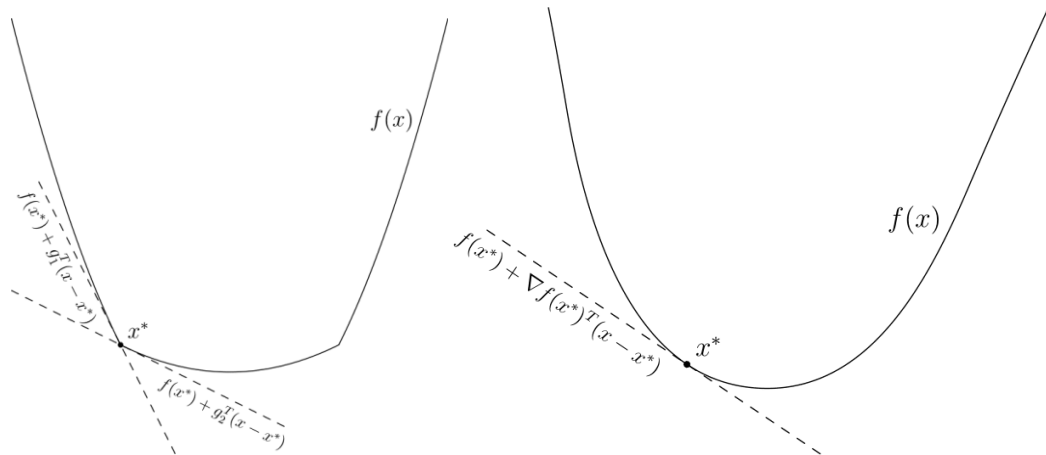


Рисунок 6.11 – Приклад субградієнту функції

Обчислити хоча б один субградієнт для точки зазвичай не дуже складно, субградієнтний спуск по суті використовує субградієнт замість градієнта. Виявляється – цього достатньо, теоретично швидкість збіжності при цьому падає, проте наприклад у нейронних мережах недиференційовану функцію $ReLU(x) = \max(0, x)$ люблять використовувати саме через те, що з нею навчання проходить швидше (це до речі приклад невикуклої недиференційованої функції, в якій успішно застосовується (суб)градієнтний спуск. Сама по собі функція ReLU опукла, але багатошарова нейронна мережа, що містить ReLU, неопукла і недиференційована). Як приклад, для функції $f(x) = |x|$ субдиференціал обчислюється дуже просто

$$\partial f(x) = \begin{cases} 1, x > 0 \\ -1, x < 0 \\ [-1, 1], x = 0 \end{cases}$$

Мабуть, остання важлива річ, яку варто знати, це те, що субградієнтний спуск не сходиться при постійному розмірі кроку. Це

найпростіше побачити для зазначеної вище функції $f(x)=|x|$. Навіть відсутність похідної в одній точці ламає збіжність:

1. Допустимо ми почали з точки x_0 .
2. Крок субградієнтного спуску:

$$x_{k+1} = \begin{cases} x_k - 1, & x > 0 \\ x_k + 1, & x < 0 \\ ???, & x = 0 \end{cases}$$

3. Якщо $x_0 > 0$, то перші кілька кроків ми будемо віднімати одиницю, якщо $x_0 < 0$, то додавати. Так чи інакше ми в якийсь момент опинимося на інтервалі $[0,1)$, з якого потрапимо до $[-1,0)$, а далі стрибатимемо між двома точками цих інтервалів.

Теоретично для субградієнтного спуску рекомендується брати послідовність кроків

$$\alpha_k = \frac{1}{(k+1)^c}$$

де c зазвичай 1 або 0.5. Насправді часто успішне застосування має крок $\alpha_k = e^{-ck}$, хоча й для таких кроків взагалі не буде збіжності.

Proximal методи

Proximal-методи постали як узагальнення проєктивних градієнтних методів. Ідея дуже проста: якщо є функція f , представлена у вигляді суми $f(x) = \varphi(x) + h(x)$, де φ – опукла функція, що диференціюється, а $h(x)$ – опукла, для якої існує спеціальний proximal-оператор $prox_h(x)$, то властивості збіжності градієнтного спуску для φ залишаються і для градієнтного спуску для f якщо після кожної ітерації застосовувати цей proximal-оператор для поточної точки x_k , тобто загальний вигляд proximal-метода виглядає так:

$$x_{k+1} = prox_{\alpha_k}(x_k - \alpha_k \nabla \varphi(x_k))$$

Поки що зовсім незрозуміло, навіщо це може знадобитися, особливо з огляду на те, що не зрозуміло, що таке проксимальний-оператор. Ось два приклади:

1. $h(x)$ – індикатор-функція опуклої множини K , тобто

$$h(x) = \begin{cases} 0, x \in K \\ +\infty, x \notin K \end{cases}$$

У цьому випадку $\text{prox}_{\text{akh}}(x)$ – це проекція на множину K , тобто «найближча до x точка множини K ». Таким чином, ми обмежуємо градієнтний спуск тільки на множину K , що дозволяє вирішувати завдання з обмеженнями. На жаль, обчислення проекції у загальному випадку може бути ще складнішим завданням, тому зазвичай такий метод застосовується, якщо обмеження мають простий вигляд, наприклад так звані **box-обмеження**: за кожною координатою

$$l_i \leq x_i \leq r_i$$

2. $h(x) = \lambda \|x\|_1 = \lambda \sum_{i=1}^n |x_i|$ – ℓ_1 -регуляризація. Такий доданок люблять додавати у завдання оптимізації в машинному навчанні, щоб уникнути перенавчання. Регуляризація такого виду ще має тенденцію обнулювати найменш значущі компоненти. Для такої функції proximal -оператор має вигляд (нижче описано вираз для однієї координати):

$$[\text{prox}_{\text{ah}}(x)]_i = \begin{cases} x_i - \alpha, x_i > \alpha \\ x_i + \alpha, x_i < -\alpha \\ 0, x_i \in [-\alpha, \alpha] \end{cases}$$

що досить просто вирахувати.

Алгоритми градієнтної оптимізації

Адаград

Adagrad – це алгоритм для **градієнтної оптимізації**, який адаптує швидкість навчання до параметрів, виконуючи менші оновлення (тобто низькі швидкості навчання) для параметрів, пов'язаних з найпоширенішими ознаками, і більші оновлення (тобто з високим рівнем навчання) для параметрів, пов'язаних з нечастими ознаками. Тому він добре підходить для роботи з розрідженими даними.

Якщо на етапі оновлення ми розділимо поточний градієнт на дуже невелику суму квадратів градієнтів, результат цього поділу стає дуже високим і навпаки, для іншої осі з високими значеннями градієнта.

В результаті ми змушуємо алгоритм робити оновлення у будь-якому напрямку з однаковими пропорціями.

Це означає, що ми прискорюємо процес оновлення вздовж осі з невеликими градієнтами, збільшуючи градієнт уздовж осі. З іншого боку, поновлення вздовж осі з великим градієнтом трохи сповільнюються.

Однак є проблема із цим алгоритмом оптимізації. Уявіть, що станеться із сумою квадратів градієнтів, якщо тренування займає багато часу. Згодом цей термін побільшає. Якщо поточний градієнт ділиться на це велике число, крок оновлення для ваги стає дуже маленьким. Це ніби ми використовували дуже низьке навчання, яке стає тим нижчим, чим довше йде тренування. У гіршому випадку ми застрягли б з AdaGrad, і навчання тривало б завжди.

Adagrad значно покращив надійність СГС і використав його для навчання великомасштабних нейронних мереж у Google.

Отже, раніше ми оновлювали всі параметри θ одразу, кожен параметр θ_i використовував ту ж швидкість навчання η . Оскільки Adagrad використовує різну швидкість навчання для кожного параметра θ_i на кожному часовому кроці t , давайте спочатку покажемо оновлення кожного параметра Adagrad, яке потім векторизуємо. Для стислості ми використовуємо g_t для позначення градієнта на часовому кроці t . $g_{t,i}$ є частковою похідною цільової функції за параметром θ_i на часовому кроці t :

$$g_{t,i} = \nabla_{\theta} J(\theta_{t,i})$$

Оновлення СГС для кожного параметра θ_i на кожному часовому кроці t потім стає:

$$\theta_{t+1,i} = \theta_{t,i} - \eta g_{t,i}$$

У своєму правилі оновлення Adagrad змінює загальну швидкість навчання η на кожному часовому кроці t для кожного параметра θ_i на основі минулих градієнтів, обчислених для θ_i :

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

$G_t \in R^{d \times d}$ – це діагональна матриця, де кожен діагональний елемент i, i – це сума квадратів градієнтів відносно θ_i з точністю до кроку t , а ϵ – це член, що згладжує, уникає поділу на нуль (зазвичай порядку $1e-8$). Цікаво, що без операції квадратного кореня алгоритм працює набагато гірше.

Оскільки G_t містить суму квадратів минулих градієнтів за всіма параметрами θ вздовж його діагоналі, тепер ми можемо векторизувати реалізацію, виконавши добуток матриця-вектор $\odot$ між G_t і g_t :

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

Однією з основних переваг Adagrad є те, що він усуває необхідність вручну налаштовувати швидкість навчання. Більшість реалізацій використовують значення за замовчуванням 0.01 і залишають його на цьому рівні.

Основним недоліком Adagrad, як вже сказано, є накопичення квадратів градієнтів у знаменнику: оскільки кожен доданий член позитивний, накопичена сума продовжує зростати під час навчання. Це, у свою чергу, призводить до скорочення швидкості навчання і, зрештою, до нескінченно малої, після чого алгоритм більше не може набувати додаткових знань. Наступні алгоритми спрямовані на усунення цього недоліку.

Ададелта

Adadelata є розширенням Adagrad, яке прагне знизити його агресивну, швидкість навчання, що монотонно зменшується. Замість того, щоб накопичувати всі попередні квадрати градієнтів, Adadelata обмежує вікно накопичених попередніх градієнтів деяким фіксованим розміром w .

Замість неефективного зберігання w для попередніх квадратів градієнтів сума градієнтів рекурсивно визначається як спадне середнє всіх минулих квадратів градієнтів (decaying average). The running average (поточне середнє)

$$E[g^2]_t$$

на часовому кроці t залежить тільки від попереднього середнього та поточного градієнта:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma)g_t^2$$

Встановлюємо γ до значення, аналогічного значенню імпульсу, близько 0.9. Для ясності тепер ми переписуємо оновлення СГС у термінах вектора оновлення параметрів $\Delta\theta_t$:

$$\Delta\theta_t = -\eta g_{t,i}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Таким чином, отриманий раніше вектор оновлення параметрів Adagrad набуває форми:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

Тепер просто замінімо діагональну матрицю $\mathbf{G}_t$ із згасаючим середнім за минулими квадратами градієнтів $E[g^2]_t$:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Оскільки знаменник – це просто критерій середньоквадратичної помилки градієнта, ми можемо замінити його скороченим критерієм:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

Автори відзначають, що одиниці виміру в цьому оновленні (а також у SGD, Momentum або Adagrad) не збігаються, тобто в оновленні повинні бути ті самі гіпотетичні одиниці, що й у параметрі. Щоб зрозуміти це, вони спочатку визначають інше експоненційно спадне середнє, цього разу не квадратів градієнтів, а квадратів оновлень параметрів:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1 - \gamma)\theta_t^2$$

Таким чином, середньоквадратична помилка оновлення параметрів:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

оскільки $RMS[\Delta\theta]_t$ невідомо, ми апроксимуємо його за допомогою RMS оновлень параметрів до попереднього тимчасового кроку. Заміна швидкості навчання η у попередньому правилі оновлення на $RMS[\Delta\theta]_{t-1}$ нарешті, дає остаточне правило оновлення параметрів Adadelta:

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

З Adadelta нам навіть не потрібно встановлювати швидкість навчання за замовчуванням, оскільки вона виключена з правила оновлення.

RMSprop

RMSprop – це неопублікований метод адаптивної швидкості навчання, запропонований Джеффом Хінтоном.

RMSprop і Adadelta були розроблені незалежно приблизно в один і той же час, що пов'язано з необхідністю вирішити проблему радикального зниження швидкості навчання Adagrad. Фактично, RMSprop ідентичний першому вектору оновлення Adadelta, який ми отримали вище:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

RMSprop також ділить швидкість навчання на експоненційно спадне середнє квадратів градієнтів. Хінтон пропонує γ має бути встановлене на 0.9, у той час як хороше значення за умовчанням для швидкості навчання η становить 0.001.

Адам

Адаптивна оцінка моменту (Адам) – ще один метод, який обчислює швидкість адаптивного навчання кожного параметра. Він, мабуть, найбільш поширеніший на даний момент метод оптимізації, **та його можна та треба використовувати з дефолтними параметрами, якщо ви не знаєте з чого почати.**

На додаток до зберігання експоненційно спадного середнього минулих квадратів градієнтів v_t як у Adadelta і RMSprop, Адам також зберігає експоненційно спадне середнє минулих градієнтів m_t , аналогічно імпульсу. У той час як імпульс можна розглядати як кулю, що біжить по схилу, Адам поводить себе як важка куля з тертям, яка, таким чином, віддає перевагу плоским мінімумам на поверхні помилки. Ми обчислюємо загасаючі середні минулих та минулих квадратів градієнтів m_t та v_t відповідно наступним чином:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

m_t та v_t є оцінками першого моменту (середнього) та другого моменту (нецентрована дисперсія) градієнтів відповідно, звідси і назва методу. Так як m_t та v_t ініціалізуються як вектори нулів, автори Адама зауважують, що вони зміщені у бік нуля, особливо на початкових тимчасових кроках, і особливо коли швидкості розпаду малі (тобто β_1 і β_2 близькі до 1).

Вони протидіють цим зсувам, обчислюючи скориговані зміщення оцінок першого та другого моментів:

$$\widehat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\widehat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Потім вони використовують їх для оновлення параметрів, як в Adadelta та RMSprop, що дає правило оновлення Адама:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\widehat{v}_t} + \epsilon} \widehat{m}_t$$

Автори пропонують наступні значення за умовчанням 0.9 для β_1 , 0.999 для β_2 і 10^{-8} для ϵ . Вони емпірично показують, що Адам добре працює на практиці та вигідно відрізняється від інших алгоритмів адаптивного методу навчання.

Інші методи

Також існують подальші модифікації методів оптимізації, наприклад AdaMax, Nadam, AMSGrad, AdamW, який фіксує зниження ваги у Адама, QNAdam, який усереднює стандартний крок SGD з кроком SGD за імпульсом, AggMo, який об'єднує кілька членів імпульсу γ , та інші [26-33].

Візуалізація алгоритмів оптимізації

Рисунок 6.12 надає деяке уявлення щодо поведінки алгоритмів оптимізації більшості представлених методів оптимізації.

На рисунку 6.12 бачимо їх поведінку на контурах поверхні втрат (зліва, Beale function). Зверніть увагу, що Adagrad, Adadelata і RMSprop майже відразу ж прямують у правильному напрямку і сходяться швидко, в той час як Momentum і NAG ведуть убік, викликаючи образ м'яча, що котиться з пагорба. NAG, однак, може швидко скоригувати свій курс завдяки своїй підвищеній чуйності, заглядаючи наперед і зводячи курс до мінімуму.

Рисунок 6.12 (правий) показує поведінку алгоритмів у сідловій точці, тобто в точці, де один вимір має позитивний нахил, а інший вимір має негативний нахил, що, як ми згадували раніше, створює труднощі для SGD. SGD, Momentum і NAG не можуть порушити симетрію, хоча двом останнім в кінцевому підсумку вдається уникнути сідлової точки, в той час як Adagrad, RMSprop і Adadelata швидко спускаються за негативним схилом.

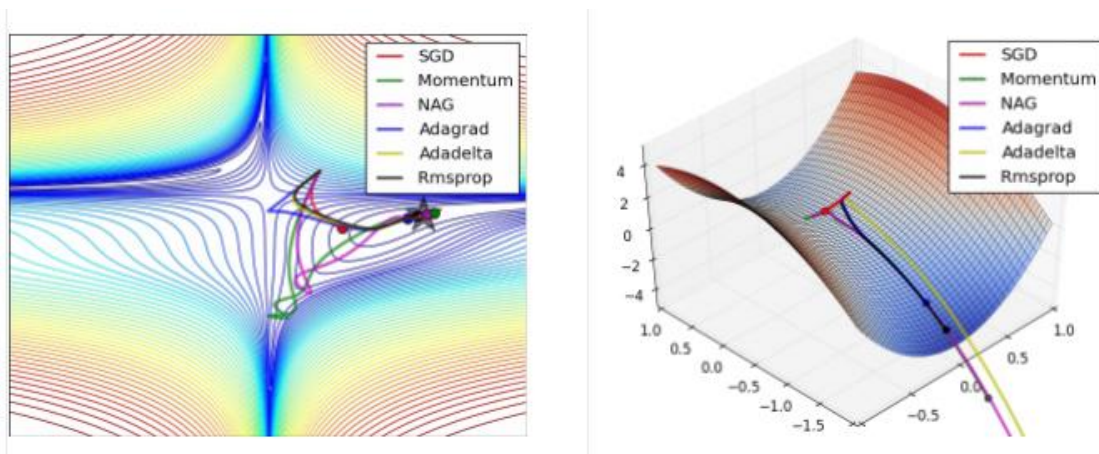


Рисунок 6.12 – Приклад поведінки різних модифікацій градієнтного спуску

Як бачимо, методи адаптивної швидкості навчання, тобто Adagrad, Adadelata, RMSprop і Adam, є найбільш підходящими й забезпечують найкращу збіжність цих сценаріїв.

Який оптимізатор використати?

Отже, який оптимізатор слід використовувати? Якщо ваші вхідні дані нечисленні, то, ймовірно, ви досягнете найкращих результатів, використовуючи один із методів адаптивної швидкості навчання. Додатковою перевагою є те, що вам не потрібно налаштовувати швидкість навчання, але ви, ймовірно, досягнете кращих результатів зі значенням за промовчанням.

Таким чином, підсумовуючи

- RMSprop – це розширення Adagrad, яке має справу з швидкістю навчання, що радикально знижується.
- RMSprop ідентичний Adadelata, за винятком того, що Adadelata використовує RMS оновлення параметрів у правилі оновлення.
- Адам додає корекцію зміщення та імпульс RMSprop.
- RMSprop, Adadelata та Adam – дуже схожі алгоритми, які добре почувуються в аналогічних обставинах.
- Корекція зміщення допомагає Адаму трохи перевершити RMSprop до кінця оптимізації, оскільки градієнти стають більш розрідженими.
- У цьому плані Адам може бути найкращим вибором.

Цікаво, що багато недавніх статей використовують SGD без імпульсу. Як було показано, SGD зазвичай вдається знайти мінімум, але це може зайняти значно більше часу. Отже, якщо ви дбаєте про швидку збіжність і тренуєте глибоку або складну нейронну мережу, вам слід вибрати один із методів адаптивної швидкості навчання.

Використання генетичного алгоритму для оптимізації рекурентних нейронних мереж (базовий огляд методу на прикладі).

Останнім часом було виконано велику роботу з автоматизації машинного навчання, від вибору відповідного алгоритму до вибору функцій та налаштування гіперпараметрів. Доступні кілька інструментів, які можуть допомогти користувачеві в процесі ефективного виконання сотень експериментів. Так само архітектура глибокої нейронної мережі зазвичай розробляється експертами; методом проб та помилок. Хоча цей підхід спричинив створення сучасних моделей у кількох областях, але потребує дуже багато часу. Останнім часом через збільшення доступної обчислювальної потужності дослідники використовують навчання з підкріпленням та еволюційні алгоритми для автоматичного пошуку оптимальних нейронних архітектур.

У цьому розділі ми побачимо, як застосувати генетичний алгоритм (GA) для пошуку оптимального розміру вікна та кількості одиниць у рекурентній нейронній мережі (RNN, архітектура буде детально розглядатися у наступних розділах) на основі довгострокової короткострокової пам'яті (LSTM). Для прикладу ми будемо навчати та оцінювати моделі для завдання прогнозування часових рядів за допомогою фреймворку **Keras**. Для GA буде використовуватися пакет Python під назвою DEAP. Основна ідея – познайомити Вас з використанням GA, автоматично підібрати оптимальні налаштування, отже, будуть досліджені лише два параметри.

Посилання на ipython netbook: <https://github.com/aqibsaeed/Genetic-Algorithm-RNN>.

Генетичний алгоритм

Генетичний алгоритм – це евристичний пошук і метод оптимізації, натхненний процесом природного відбору. Він широко використовується для пошуку оптимального розв'язання задач оптимізації з великим простором параметрів. Процес еволюції видів (рішення у нашому випадку)

імітується залежно від біологічно натхнених компонентів, наприклад, кросовера. Крім того, оскільки він не бере до уваги допоміжну інформацію (наприклад, похідні), його можна використовувати як для дискретної, так і безперервної оптимізації (рисунок 6.13).

Для використання GA повинні бути виконані дві попередні умови: а) подання рішення або визначення хромосоми; б) функція пристосованості (fitness function) для оцінки отриманих рішень. У нашому випадку двійковий масив є генетичним уявленням рішення, а середньоквадратична помилка моделі (RMSE) на наборі перевірки буде діяти як значення придатності. Більш того, три основні операції, що становлять GA, такі:

1. **Вибір:** визначає, які рішення зберегти для подальшого відтворення, наприклад, вибір колеса рулетки.
2. **Кросовер:** описує, як нові рішення створюються з існуючих, наприклад, n -точковий кросовер.
3. **Мутація:** його мета полягає в тому, щоб внести різноманітність і новизну в пул рішень за допомогою випадкової заміни або відключення бітів рішення, наприклад, бінарної мутації.

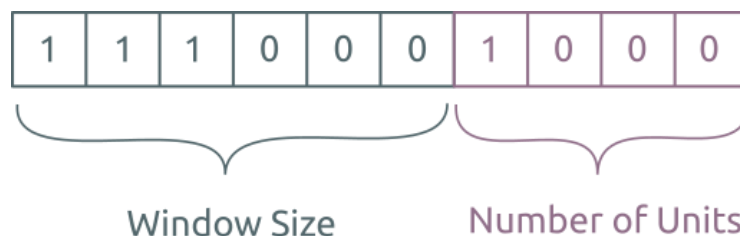


Рисунок 6.13 – Генетичне представлення рішення

Іноді також використовується метод, який називається «елітарністю», який зберігає кілька найкращих рішень від населення і передає їх наступному поколінню. На рисунку 6.14 зображено повний генетичний алгоритм, у якому випадково генеруються початкові рішення (популяція). Потім вони оцінюються відповідно до функції пристосованості (fitness function), після чого виконуються відбір, кросовер та мутація. Цей процес повторюється протягом певної кількості ітерацій (званих поколіннями у

термінології GA). Зрештою, рішення з найвищою оцінкою придатності вибирається як найкраще рішення.

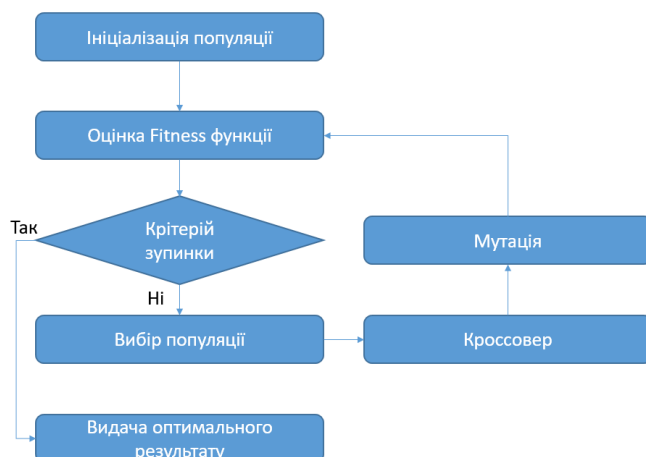


Рисунок 6.14 – Повний цикл генетичного алгоритму

Підсумовуючи, як працюють генетичні алгоритми.

За своєю суттю генетичний алгоритм:

1. Створює популяцію із (випадково згенерованих) членів
2. Оцінює кожного члена населення з урахуванням певної мети. Ця оцінка називається фітнес-функцією.
3. Відбирає та розводить кращих представників популяції, щоб виробляти більше схожих на них.
4. Випадковим чином мутує деяких членів, щоб спробувати знайти ще найкращих кандидатів
5. Вбиває інших (виживання найсильніших і всіх)
6. Процес повторюється починаючи з кроку 2. Кожна ітерація цих кроків називається поколінням.

Повторюємо цю процедуру достатньо разів, і повинні залишитися тільки найкращі можливі члени населення.

Реалізація генетичного алгоритму на прикладі RNN моделі

Тепер ми маємо чітке уявлення про те, що таке GA і як він працює. Далі перейдемо до прикладу коду.

Ми будемо використовувати дані прогнозу енергії вітру, які доступні за наступним посиланням <https://www.kaggle.com/c/GEF2012-wind-forecasting/data>. Він складається з нормалізованих (від нуля до одиниці) вимірів енергії вітру від семи вітрових електростанцій. Для простоти ми будемо використовувати дані першої вітряної електростанції (стовпець з ім'ям **wp1**), але спробуйте проєкспериментувати та розширити код для прогнозування енергії для всіх семи вітрових електростанцій самостійно.

Спершу імпортуємо необхідні пакети, завантажимо набір даних та визначимо дві допоміжні функції. Перший метод *prepare_dataset* сегментує дані на частини для створення пари X , Y для навчання моделі. X – це значення потужності вітру з минулого та Y – це значення в майбутньому в момент часу t . Другий метод *train_evaluate* виконує три операції: 1) декодування рішення GA для отримання розміру вікна та кількості одиниць. 2) Підготовка набору даних, використовуючи розмір вікна, знайдений GA, і поділ на набір для навчання та перевірки, і 3) навчання моделі LSTM, обчислення RMSE на наборі перевірки та повернення його як оцінку придатності поточного рішення GA.

Імпортування бібліотек:

```
import numpy as np
import pandas as pd
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split as split

from keras.layers import LSTM, Input, Dense
from keras.models import Model

from deap import base, creator, tools, algorithms
from scipy.stats import bernoulli
from bitstring import BitArray
```

Завантаження даних (функція *prepare_dataset*):

```
np.random.seed(1120)

data = pd.read_csv('train.csv')
```

```

data = np.reshape(np.array(data['wp1']), (len(data['wp1']), 1))

# Use first 17,257 points as training/validation and rest of the 1500 points
as test set.
train_data = data[0:17257]
test_data = data[17257:]
def prepare_dataset(data, window_size):
    X, Y = np.empty((0, window_size)), np.empty((0))
    for i in range(len(data)-window_size-1):
        X = np.vstack([X, data[i:(i + window_size), 0]])
        Y = np.append(Y, data[i + window_size, 0])
    X = np.reshape(X, (len(X), window_size, 1))
    Y = np.reshape(Y, (len(Y), 1))
    return X, Y

```

Оцінка та тренування моделі:

```

def train_evaluate(ga_individual_solution):
    # Decode GA solution to integer for window_size and num_units
    window_size_bits = BitArray(ga_individual_solution[0:6])
    num_units_bits = BitArray(ga_individual_solution[6:])
    window_size = window_size_bits.uint
    num_units = num_units_bits.uint
    print('\nWindow Size: ', window_size, ', Num of Units: ', num_units)

    # Return fitness score of 100 if window_size or num_unit is zero
    if window_size == 0 or num_units == 0:
        return 100,

    # Segment the train_data based on new window_size; split into train
    and validation (80/20)
    X, Y = prepare_dataset(train_data, window_size)
    X_train, X_val, y_train, y_val = split(X, Y, test_size = 0.20,
    random_state = 1120)

    # Train LSTM model and predict on validation set
    inputs = Input(shape=(window_size, 1))
    x = LSTM(num_units, input_shape=(window_size, 1))(inputs)
    predictions = Dense(1, activation='linear')(x)
    model = Model(inputs=inputs, outputs=predictions)
    model.compile(optimizer='adam', loss='mean_squared_error')
    model.fit(X_train, y_train, epochs=5, batch_size=10, shuffle=True)
    y_pred = model.predict(X_val)

```

```

# Calculate the RMSE score as fitness score for GA
rmse = np.sqrt(mean_squared_error(y_val, y_pred))
print('Validation RMSE: ', rmse, '\n')

return rmse

```

Потім використовуємо пакет DEAP, щоб визначити змінні для запуску GA. Ми будемо використовувати двійкове представлення для рішення довжини десять. Воно буде ініціалізовано випадково з використанням розподілу Бернуллі. Аналогічним чином використовується *crossover*, *shuffle mutation* та *roulette wheel selection*. Значення параметрів GA ініціалізуються довільно (спробуйте дослідити різні налаштування).

```

population_size = 4
num_generations = 4
gene_length = 10

# As we are trying to minimize the RMSE score, that's why using -1.0.
# In case, when you want to maximize accuracy for instance, use 1.0
creator.create('FitnessMax', base.Fitness, weights = (-1.0,))
creator.create('Individual', list, fitness = creator.FitnessMax)

toolbox = base.Toolbox()
toolbox.register('binary', bernoulli.rvs, 0.5)
toolbox.register('individual', tools.initRepeat, creator.Individual,
toolbox.binary,
n = gene_length)
toolbox.register('population', tools.initRepeat, list, toolbox.individual)

toolbox.register('mate', tools.cxOrdered)
toolbox.register('mutate', tools.mutShuffleIndexes, indpb = 0.6)
toolbox.register('select', tools.selRoulette)
toolbox.register('evaluate', train_evaluate)

population = toolbox.population(n = population_size)
r = algorithms.eaSimple(population, toolbox, cxpb = 0.4, mutpb = 0.1,
ngen = num_generations, verbose = False)

```

Найкращі рішення, знайдені через GA, можна легко побачити, використовуючи `tools.selBest(population, k=1)`. Після цього оптимальну

конфігурацію можна використовувати для навчання на повному навчальному наборі та тестування його на тестовому наборі.

```
# Print top N solutions - (1st only, for now)
best_individuals = tools.selBest(population,k = 1)
best_window_size = None
best_num_units = None

for bi in best_individuals:
    window_size_bits = BitArray(bi[0:6])
    num_units_bits = BitArray(bi[6:])
    best_window_size = window_size_bits.uint
    best_num_units = num_units_bits.uint
    print('\nWindow Size: ', best_window_size, ', Num of Units: ',
best_num_units)

# Train the model using best configuration on complete training set
#and make predictions on the test set
X_train,y_train = prepare_dataset(train_data,best_window_size)
X_test,y_test = prepare_dataset(test_data,best_window_size)

inputs = Input(shape=(best_window_size,1))
x = LSTM(best_num_units, input_shape=(best_window_size,1))(inputs)
predictions = Dense(1, activation='linear')(x)
model = Model(inputs = inputs, outputs = predictions)
model.compile(optimizer='adam',loss='mean_squared_error')
model.fit(X_train, y_train, epochs=5, batch_size=10,shuffle=True)
y_pred = model.predict(X_test)

rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print('Test RMSE: ', rmse)
```

Таким чином ми побачили, як використовувати ГА для автоматичного пошуку оптимального розміру вікна (або ретроспективного аналізу) і кількості одиниць для використання в RNN. Для подальшого навчання треба поекспериментувати з різними конфігураціями параметрів ГА, розширити генетичне уявлення, щоб увімкнути більше параметрів для вивчення.

Оптимізація повнозв'язкової нейронної мережі (MLP) генетичним алгоритмом.

Спробуємо розробити повнозв'язну нейронну мережу (MLP). Наша мета – знайти найкращі параметри для завдання класифікації зображень.

Налаштуємо чотири параметри:

- Кількість шарів (або глибина мережі)
- Кількість нейронів на шар (або ширину мережі)
- Функція активації щільного шару
- Оптимізатор мережі

Кроки, які ми зробимо для розвитку мережі, аналогічні тим, що описані вище:

1. Ініціалізуємо N випадкових мереж, щоб створити нашу популяцію.

2. Оцінимо кожен мережу. Це займе деякий час: ми повинні навчити ваги кожної мережі, а потім подивитися, як добре вона працює при класифікації на тестовому набору. Оскільки це буде завдання класифікації зображень, ми будемо використовувати точність класифікації як нашу функцію придатності.

3. Відсортуюмо всі мережі в нашій популяції за балом (точністю). Збережемо частину кращих мереж, щоб вони стали частиною наступного покоління та розводили дітей.

4. Випадково збережемо кілька мереж, що не входять до топу. Це допомагає знаходити потенційно вдалі комбінації між гіршими та кращими виконавцями, а також допомагає нам не застрягати у локальному максимумі.

5. Тепер, коли ми вирішили, які мережі залишити, ми випадково змінюємо деякі параметри в деяких мережах.

6. І найцікавіше: припустимо, ми почали з популяції з 20 мереж, ми зберегли верхні 25% (5 мереж), випадковим чином залишили ще 3 мережі, що програли, і змінили кілька з них. Ми дозволили іншим 12 мережам

померти. Щоб зберегти населення у 20 мережах, нам потрібно заповнити 12 відкритих місць. Час розводити!

Розведення

Розведення – це коли ми беремо двох членів популяції і генеруємо одного або кількох дітей, де ця дитина є комбінацією своїх батьків.

У нашому випадку нейронної мережі кожен дочірній елемент є комбінацією випадкового набору параметрів від своїх батьків. Наприклад, один дочірній елемент може мати таку кількість шарів, що і його мати, а інші параметри – від батька. У другій дитини тих самих батьків може бути протилежне. Ви можете побачити, як це відображає реальну біологію та як це може швидко призвести до оптимізації мережі.

Набір даних

Будемо використовувати відносно простий, але складний набір даних CIFAR10 (<https://www.cs.toronto.edu/~kriz/cifar.html>) для нашого експерименту. Цей набір даних ставить досить складне завдання, тому більшість параметрів не підійде, і в той же час досить простий, щоб MLP отримав гідну оцінку точності.

Примітка. Згортова нейронна мережа, безумовно, є найкращим вибором для завдання класифікації зображень 10 класів, такої як CIFAR10. Але повнозв'язна мережа чудово підійде для ілюстрації ефективності використання генетичного алгоритму для налаштування гіперпараметрів.

Код програми

Код знаходиться за цим посиланням: <https://github.com/harvitronix/neural-network-genetic-algorithm>. Це частини `optimizer.py` модуля, що містять основу коду генетичного алгоритму

```
def create_population(self, count):  
    """Create a population of random networks.  
    Args:  
        count (int): Number of networks to generate, aka the  
            size of the population
```

```

"""
pop = []
for _ in range(0, count):
    # Create a random network.
    network = Network(self.nn_param_choices)
    network.create_random()

    # Add the network to our population.
    pop.append(network)

return pop

```

Почнемо із створення популяції. Це створює екземпляри «рахункових» мереж з випадково ініціалізованими налаштуваннями та додає їх до нашого «спливаючого» списку. Це насіння для всіх поколінь.

```

def breed(self, mother, father):
    """Make two children as parts of their parents.
    Args:
        mother (dict): Network parameters
        father (dict): Network parameters
    """
    children = []
    for _ in range(2):

        child = {}

        # Loop through the parameters and pick params for the kid.
        for param in self.nn_param_choices:
            child[param] = random.choice(
                [mother.network[param], father.network[param]]
            )

        # Now create a network object.
        network = Network(self.nn_param_choices)
        network.create_set(child)

        children.append(network)

    return children

```

Розведення дітей з наших найбільш пристосованих мереж – ось де відбувається найбільша частина «чаклунства». Селекція буде різною для кожного застосування генетичних алгоритмів, і в нашому випадку ми вирішили випадково вибрати параметри для дитини від матері та батька.

```
def mutate(self, network):
    """Randomly mutate one part of the network.
    Args:
        network (dict): The network parameters to mutate
    """
    # Choose a random key.
    mutation = random.choice(list(self.nn_param_choices.keys()))

    # Mutate one of the params.
    network.network[mutation] =
    random.choice(self.nn_param_choices[mutation])

    return network
```

Мутація також дуже важлива, оскільки допомагає нам здобути випадкову велич. У нашому випадку ми випадково вибираємо параметр, а потім випадково вибираємо для нього новий параметр. Насправді він може мутувати в те саме, але це все удача.

```
def evolve(self, pop):
    """Evolve a population of networks.
    Args:
        pop (list): A list of network parameters
    """
    # Get scores for each network.
    graded = [(self.fitness(network), network) for network in pop]

    # Sort on the scores.
    graded = [x[1] for x in sorted(graded, key=lambda x: x[0],
reverse=True)]

    # Get the number we want to keep for the next gen.
    retain_length = int(len(graded)*self.retain)
```



```

# The parents are every network we want to keep.
parents = graded[:retain_length]

# For those we aren't keeping, randomly keep some anyway.
for individual in graded[retain_length:]:
    if self.random_select > random.random():
        parents.append(individual)

# Randomly mutate some of the networks we're keeping.
for individual in parents:
    if self.mutate_chance > random.random():
        individual = self.mutate(individual)

# Now find out how many spots we have left to fill.
parents_length = len(parents)
desired_length = len(pop) - parents_length
children = []

# Add children, which are bred from two remaining networks.
while len(children) < desired_length:

    # Get a random mom and dad.
    male = random.randint(0, parents_length-1)
    female = random.randint(0, parents_length-1)

    # Assuming they aren't the same network...
    if male != female:
        male = parents[male]
        female = parents[female]

    # Breed them.
    babies = self.breed(male, female)

    # Add the children one at a time.
    for baby in babies:
        # Don't grow larger than desired length.
        if len(children) < desired_length:
            children.append(baby)

parents.extend(children)

return parents

```

У цьому *evolve* методі все пов'язано воєдино. Кожен запуск цього методу є окремою еволюцією.

Отримані результати

Ми почнемо із запуску алгоритму ***brute force***, щоб знайти найкращу мережу. Тобто ми навчимо та протестуємо всі можливі комбінації наданих нами мережевих параметрів.

Наш найкращий результат з використанням *brute force* може бути наступним: точність 56,03% за наступних параметрів:

- Шари: 2
- Нейрони: 768
- Активація: elu
- Оптимізатор: adamax

Це може зайняти більше 60 годин.

Тепер ми запустимо наш генетичний алгоритм, починаючи з популяції з 20 випадково ініціалізованих мереж і запустимо його протягом 10 поколінь.

Фінальний рахунок може бути: 56,56% точність, більше семи годин час виконання. Мережі практично ідентичні:

- Шари: 2
- Нейрони: 512
- Активація: elu
- Оптимізатор: adamax

Єдина різниця в тому, що в генетичному алгоритмі віддається перевага від 512 до 768 нейронів. (Під час перебору мережа 512 досягла 55,65%. Повинно було бути встановлено випадкове початкове число)

Так у чому ж справа? Генетичний алгоритм дав нам той самий результат у 1/9 разу! Сім годин замість 60. Тобто, ймовірно, що зі збільшенням складності параметра генетичний алгоритм дає експоненційний виграш у швидкості.

ТЕМА №7: ЗГОРТКА. ЗГОРТКОВИЙ ШАР ТА ЗГОРТКОВА НЕЙРОННА МЕРЕЖА (CNN).

Вступ. Загальні зауваження.

Найкращі результати в області розпізнавання обличчя показала *Convolutional Neural Network* або згорткова нейронна мережа (далі – ЗНМ або CNN), яка є логічним розвитком ідей таких архітектур ШНМ, як когнітрон та неокогнітрон. Їх успіх обумовлений можливістю обліку двовимірної топології зображення, на відміну від багат шарового перцептрона [35-37].

Згорткові нейронні мережі забезпечують часткову стійкість до змін масштабу, зсувів, поворотів, зміни ракурсу та інших спотворень. Згорткові нейронні мережі поєднують три архітектурні ідеї, для забезпечення інваріантності до зміни масштабу, повороту зсуву та просторових спотворень:

- локальні рецепторні поля (забезпечують локальну двовимірну зв'язність нейронів);
- загальні синаптичні коефіцієнти (забезпечують детектування деяких рис в будь-якому місці зображення та зменшують загальну кількість вагових коефіцієнтів);
- ієрархічна організація із просторовими підвибірками.

На даний момент згорткова нейронна мережа та її модифікації вважаються найкращими за точністю та швидкістю алгоритмами знаходження об'єктів на сцені. Починаючи з 2012 року, нейромережі займають перші місця на відомому міжнародному конкурсі з розпізнавання образів ImageNet.

Згорткові нейронні мережі (CNN) призначені для відображення даних зображення (або двовимірних багатовимірних даних) у вихідну змінну (одномірні дані). Вони виявилися настільки ефективними, що є готовим до

використання методом для будь-якого типу завдань прогнозування з використанням даних зображення як вхідні дані (рисунок 7.1).

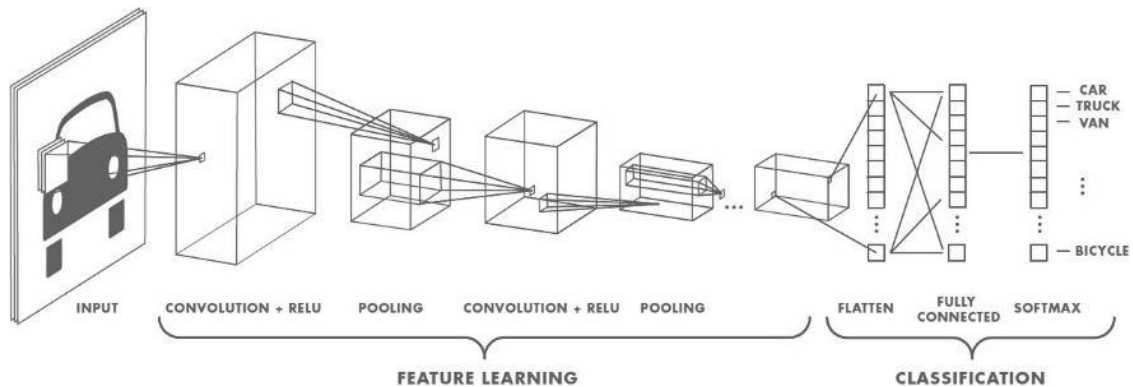


Рисунок 7.1 – Приклад багатoshарової згорткової нейронної мережі для обробки зображення

Перевага використання CNN – це їхнє вміння розвивати внутрішнє уявлення двовимірного зображення. Це дозволяє моделі пізнавати положення та масштаб у різних структурах даних, що важливо при роботі із зображеннями [35-40].

Згорткову нейронну мережу можна ефективно використовувати для:

- Дані зображення
- Проблеми прогнозування класифікації
- Проблеми прогнозування регресії

Дані з просторовим взаємозв'язком

У цілому нині CNN добре працюють із даними, мають просторові відносини.

Вхід CNN традиційно є двовимірним, полем або матрицею, але також може бути змінений на одновірний, що дозволяє створювати внутрішнє уявлення одновірної послідовності.

Це дозволяє використовувати CNN у більш загальному плані для інших типів даних, що мають просторові відносини.

Наприклад:

- У текстовому документі є відносини порядку між словами

- У часових кроках часового ряду існує впорядкований взаємозв'язок.

Хоча CNN були спеціально розроблені для даних, що не є зображеннями, вони досягають найсучасніших результатів з таких проблем, як класифікація документів, що використовується в аналізі тональності та пов'язаних із ними проблем.

Згорткові нейронні мережі також можна використовувати з такими даними:

- Текстові дані
- Дані тимчасового ряду
- Вхідні дані послідовності

Топ-7 додатків згорткових нейронних мереж

Прості застосування CNN, які ми можемо бачити у повсякденному житті, є очевидним вибором, наприклад, програмне забезпечення для розпізнавання обличчя, класифікація зображень, програми розпізнавання мовлення тощо. Це терміни, з якими ми знайомі і становлять більшу частину нашого повсякденного життя. Життя, особливо у соціальних мережах, орієнтованих на зображення, таких як Instagram. Деякі з можливих практичних доданків CNN [37-40]:

1. Розшифрування розпізнавання обличчя

Розпізнавання обличчя розбивається згортковою нейронною мережею на такі основні компоненти:

- Виявлення кожного обличчя на зображенні
- Зосередження уваги на кожному обличчі незалежно від зовнішніх чинників, як-от світло, ракурс, поза.
- Виявлення унікальних особливостей
- Порівняння всіх зібраних даних з існуючими даними в базі даних, щоб зіставити обличчя з ім'ям.

Аналогічний процес виконується і для маркування сцени.

2. Аналіз документів

Згорткові нейронні мережі також можуть використовуватись для аналізу документів. Це не тільки корисно для аналізу почерку, але й має велике значення для розпізнавачів. Щоб машина могла сканувати запис людини, а потім порівнювати її з наявною у неї великою базою даних, вона повинна виконувати майже мільйон команд на хвилину. Кажуть, що з використанням CNN і нових моделей і алгоритмів частота помилок була знижена до мінімум 0.4%.

3. Історичні та екологічні колекції

CNN також використовуються для складніших цілей, таких як колекції природної історії. Ці колекції виступають як ключові учасники в документуванні основних історій, таких як біорізноманіття, еволюція, втрата довкілля, біологічне вторгнення та зміна клімату.

4. Розуміння клімату

CNN можуть бути використані для того, щоб відіграти важливу роль у боротьбі зі зміною клімату, особливо для розуміння причин, з яких ми спостерігаємо такі різкі зміни, і того, як ми могли б експериментувати, щоб приборкати цей ефект. Говорять, що дані в таких колекціях природної історії можуть також надати більш глибоке соціальне та наукове розуміння, але для цього будуть потрібні кваліфіковані людські ресурси, такі як дослідники, які можуть фізично відвідувати ці типи сховищ. Потрібно більше людей для проведення глибших експериментів у цій галузі.

5. Сірі області

Введення сірої зони у CNN покликане забезпечити більш реалістичну картину реального світу. В даний час CNN в основному функціонують так само, як машина, що бачить справжнє і хибне значення для кожного питання. Однак, як люди, ми розуміємо, що реальний світ проявляється у тисячах відтінків сірого. Якщо дозволити машині розуміти та обробляти нечітку логіку, це допоможе їй зрозуміти сіру зону, в якій ми, люди, живемо та з якою прагнемо працювати. Це допоможе CNN отримати більш цілісне уявлення, що бачить людина.

6. Реклама

CNN вже значно змінили рекламу, впровадивши програмні покупки та персоналізовану рекламу на основі даних.

7. Інші цікаві області

CNN готові стати майбутнім з їх впровадженням у безпілотні автомобілі, роботів, які можуть імітувати людську поведінку, помічників у проектах з картування генному людині, прогнозування землетрусів та стихійних лих та, можливо, навіть самодіагностики медичних проблем. Таким чином, вам навіть не доведеться їхати до клініки або записуватися на прийом до лікаря, щоб переконатися, що напад чхання чи висока температура – це просто грип, а не симптоми якогось рідкісного захворювання. Одна з проблем, над якою працюють дослідники з CNN, – це виявлення раку мозку. Раніше виявлення раку мозку може виявитися великим кроком у порятунку більшої кількості життів, які постраждали від цього захворювання.

Загальна будова згорткової нейронної мережі (CNN).

У глибокому навчанні, згорткова нейронна мережа (CNN або ConvNet) є класом глибоких нейронних мереж, що найбільш широко застосовуються для аналізу візуальних образів [35-36].

CNN використовують різновид багат шарових персептронів, що вимагають мінімальної попередньої обробки. Вони також відомі як інваріантні до зсуву або просторово-інваріантні штучні нейронні мережі (SIANN), засновані на їх архітектурі з вагами, що розділяються, і характеристиках інваріантності трансляції.

Згорткові мережі були надихнуті на біологічних процесах у тому, що структура з'єднання між нейронами нагадує організацію тваринної зорової кори. Окремі нейрони кори відповідають на стимули тільки в обмеженій області зору, відомої як *рецептивне поле*. Рецептивні поля різних нейронів частково перекриваються, тому вони покривають все поле зору.

CNN використовують відносно невелику попередню обробку порівняно з іншими алгоритмами класифікації зображень. Це означає, що мережа вивчає фільтри, які у традиційних алгоритмах були створені вручну. Ця незалежність від попередніх знань та людських зусиль при розробці функцій є великою перевагою.

Вони мають додатки до зображень і відео, розпізнавання, рекомендаційних систем, класифікації зображень, аналізу медичних зображень та обробки природної мови.

Згортковий шар

Згорткові шари застосовують до входу операцію згортки, передаючи результат наступному шару. Згортка імітує реакцію окремого нейрона на зорові стимули [35-38].

Кожен згортковий нейрон обробляє дані лише для свого рецептивного поля. Хоча повнозв'язані нейронні мережі з прямим зв'язком можна використовувати вивчення функцій, і навіть для класифікації даних, застосовувати цю архітектуру до зображень недоцільно. Була б потрібна дуже велика кількість нейронів, навіть у дрібній (протилежній глибокій) архітектурі, через дуже великі вхідні розміри, пов'язані із зображеннями, де кожен піксель є релевантною змінною. Наприклад, повністю зв'язаний шар (маленького) зображення розміром 100×100 має 10000 ваг для кожного нейрона в другому шарі. Операція згортки вирішує цю проблему, оскільки зменшує кількість вільних параметрів, дозволяючи мережі бути глибшою з меншою кількістю параметрів. Наприклад, незалежно від розміру зображення, мозаїчні області розміром 5×5 , кожна з однаковою загальною вагою, потребує всього 25 параметрів, що навчаються. Таким чином, вирішується проблема градієнтів, що зникають або вибухають, при навчанні традиційних багатошарових нейронних мереж з багатьма рівнями з використанням зворотного поширення.

Об'єднання (Pooling)

Згорткові мережі можуть включати локальні або глобальні рівні об'єднання, які об'єднують виходи кластерів нейронів на одному рівні в один нейрон на наступному рівні. Наприклад, *maxpool* використовує максимальне значення кожного кластера нейронів на попередньому рівні. Інший приклад – *average Pool*, який використовує середнє значення кожного кластера нейронів на попередньому рівні.

Повнозв'язковий шар

Повністю пов'язані шари поєднують кожен нейрон одного шару з кожним нейроном іншого шару. В принципі, це те саме, що і традиційна багатошарова нейронна мережа персептрона (MLP). Згладжена матриця проходить через повністю зв'язаний шар для класифікації зображень.

Рецептивне поле

У нейронних мережах кожен нейрон отримує вхідні дані з певної кількості місць на попередньому рівні. У пов'язаному шарі кожен нейрон отримує вхідні дані від кожного елемента попереднього шару. У згортковому шарі нейрони одержують вхідні дані лише з обмеженої підобласті попереднього шару. Зазвичай підобласть має квадратну форму (наприклад розмір 5 на 5). Область введення нейрона називається його рецептивним полем. Отже, в пов'язаному шарі рецептивне поле – це весь попередній шар. У згортковому шарі сприйнятлива область менше, ніж весь попередній шар [40].

Вага

Кожен нейрон в нейронній мережі обчислює вихідне значення, застосовуючи деяку функцію до вхідних значень, що надходять з поля, що сприймає в попередньому шарі. Функція, яка застосовується до вхідних значень, визначається вектором ваги і зміщенням (зазвичай дійсними числами). Навчання в нейронній мережі прогресує за рахунок поступових коригувань зсувів та ваг. Вектор ваги та зміщення називається **фільтром** і

представляє деяку характеристику вхідних даних (наприклад, певну форму). Відмінною рисою CNN є те, що багато нейронів використовують один і той же фільтр. Це скорочує обсяг пам'яті, оскільки єдине зміщення і єдиний вектор ваг використовуються у всіх полях, що приймають, спільно використовують цей фільтр, а не в кожному приймаючому полі, що має власне зміщення і вектор ваг.

Технічно у моделі CNN з глибоким навчанням для навчання та тестування, кожне вхідне зображення проходить його через серію шарів згортки з фільтрами (*Kernels*), *Pooling*, повністю пов'язаними шарами (*FC*) і застосовуватиме функцію *Softmax* для класифікації об'єкта з імовірнісними значеннями від 0 і 1.

Як працює CNN – математичний апарат на прикладі.

Згортка

Згорткова нейронна мережа заснована на напрочуд потужній та універсальній математичній операції, яка зветься **згортка** (рисунок 7.2)

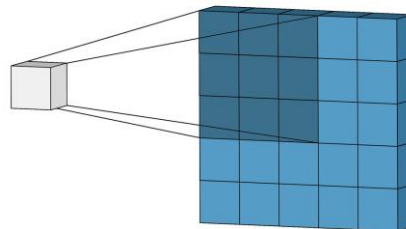


Рисунок 7.2 – Двовимірна згортка

Двовимірна згортка (2D convolution) – це досить проста операція: починаємо з ядра, що представляє собою матрицю ваг (*weight matrix*). Ядро "ковзає" над двовимірним зображенням, поелементно виконуючи операцію множення з тією частиною вхідних даних, над якою воно зараз знаходиться, а потім підсумовує всі отримані значення в один вихідний піксель [35].

Ядро повторює цю процедуру з кожною локацією, над якою воно “ковзає”, перетворюючи двовимірну матрицю на іншу двовимірну матрицю ознак. Ознаки на виході є зваженими сумами (де ваги є значеннями самого ядра) ознак на вході, розташованих приблизно в тому ж місці, що вихідний піксель на вхідному шарі (рисунок 7.3).

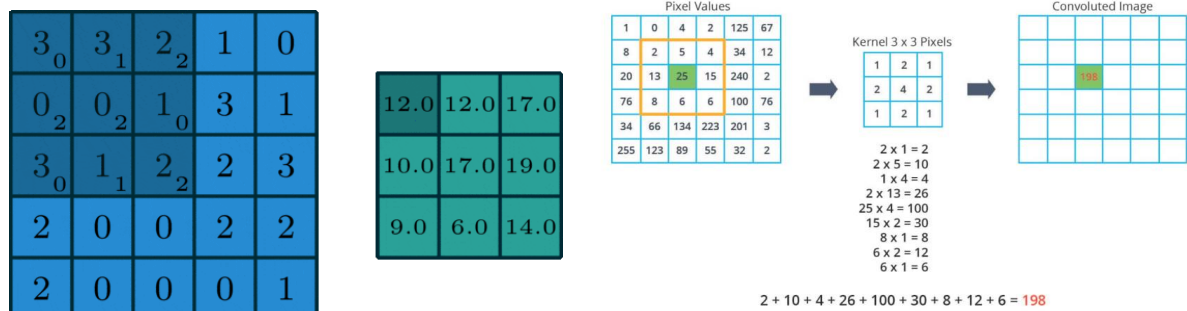


Рисунок 7.3 – Приклад операції згортки

Незалежно від того, чи потрапляє вхідна ознака в "приблизно те саме місце", вона визначається в залежності від того, знаходиться вона у зоні ядра, що створює вихідні дані, чи ні. Це означає, що розмір ядра згорткової нейронної мережі визначає кількість ознак, які будуть об'єднані для отримання нової ознаки на виході.

У наведеному вище прикладі ми маємо $5 \times 5 = 25$ ознак на вході і $3 \times 3 = 9$ ознак на виході. Для стандартного шару (*standard fully connected layer*) ми мали вагову матрицю $25 \times 9 = 225$ параметрів, а кожна вихідна ознака була зваженою сумою всіх ознак на вході. Згортка дозволяє зробити таку операцію з усього 9-ма параметрами, адже кожна ознака на виході виходить аналізом не кожної ознаки на вході, а тільки однієї вхідної, що знаходиться в "приблизно тому ж місці". Треба звернути на це увагу, оскільки це буде важливим для подальшого обговорення.

Ще приклади згорток

Отже, згортка – це перший шар для вилучення функцій із вхідного зображення. Згортка зберігає взаємозв'язок між пікселями, вивчаючи особливості зображення за допомогою невеликих областей вхідних даних.

Це математична операція, яка приймає два входи, такі як матриця зображення та фільтр або ядро.

Логічний зміст згортки такий – що більше величина елемента згортки, то більше ця частина матриці А була схожа на матрицю В (схожа у сенсі скалярного добутку). Тому матрицю А називають зображенням, а матрицю В – фільтром чи зразком (рисунок 7.4).

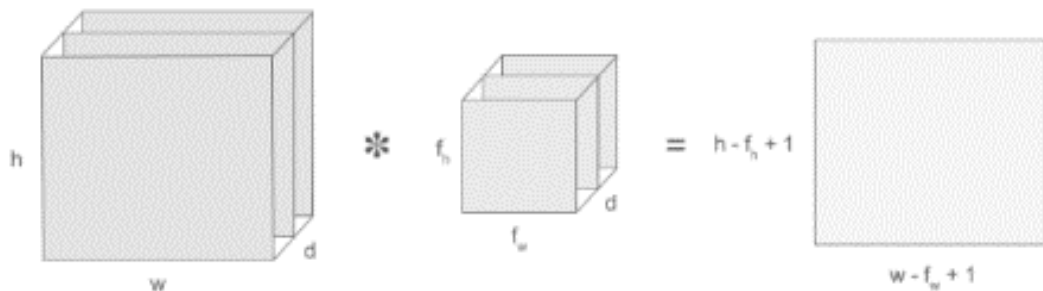


Рисунок 7.4 – Матриця зображення помножує ядро або матрицю фільтра: зображення розміром $(h \times w \times d)$, фільтр розміром $(f_h \times f_w \times d)$, на виході маємо об'єм розміром $(h-f_h+1) \times (w-f_w+1) \times 1$

Розглянемо вхідну матрицю розміром 5×5 , значення пікселів зображення якого дорівнюють 0, 1 і матрицю фільтра розміром 3×3 , як показано на рисунку 7.5.



Рисунок 7.5 – Вхідна матриця зображення (зліва) помножується на ядро або фільтр (праворуч)

Потім згортка матриці зображення 5×5 множиться на матрицю фільтра 3×3 і отримуємо вихідну мапу, яка називається "*Карта характеристик*", як показано на рисунку 7.6.

1 _{x3}	1 _{x0}	1 _{x3}	0	0
0 _{x0}	1 _{x1}	1 _{x0}	1	0
0 _{x3}	0 _{x0}	1 _{x3}	1	1
0	0	1	1	0
0	1	1	0	0

4		

Рисунок 7.6 – Вихідна матриця 3 x 3

Згортка зображення за допомогою різних фільтрів може виконувати такі операції, як виявлення країв, розмиття та підвищення різкості за допомогою фільтрів. У наведеному на рисунку 7.7 прикладі показані різні зображення згортки після застосування різних типів фільтрів (ядер).

Операція	Фільтр	Вихідне зображення
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Рисунок 7.7 – Деякі загальні фільтри

Багатоканальна версія згорткової нейронної мережі

Наведені вище діаграми стосуються лише випадку, коли зображення має один вхідний канал. На практиці більшість вхідних зображень мають 3 канали, і чим глибше ви в мережі, тим більше це число (рисунок 7.8).

У більшості випадків ми маємо справу із зображеннями RGB з трьома каналами. Ось де ключові відмінності між термінами стають потрібними: тоді як у випадку з 1 каналом терміни "фільтр" та "ядро" взаємозамінні, у загальному випадку вони різні [35-40].

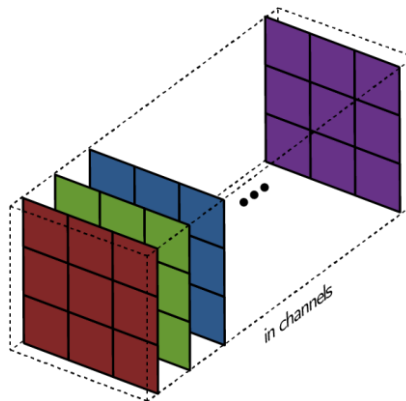


Рисунок 7.8 – Приклад багатоканального входу

Кожен фільтр насправді є колекцією ядер, причому для кожного окремого вхідного каналу цього шару є одне ядро, і кожне ядро унікальне.

Кожен фільтр у згортковому шарі створює лише один вихідний канал і роблять вони це так: кожне з ядер фільтра «ковзає» по їх відповідним вхідним каналам, створюючи оброблену версію кожного з них (рисунок 7.9). Деякі ядра можуть мати більшу вагу, ніж інші, щоб приділяти більшу увагу певним вхідним каналам (наприклад, фільтр може задати червоному каналу ядра більшу вагу, ніж іншим каналам, і, отже, більше реагувати на відмінності в образах з червоного каналу).

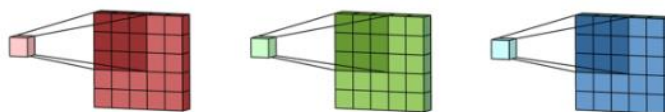


Рисунок 7.9 – Приклад сканування ядрами багатоканального входу

Потім кожна з оброблених у каналі версій підсумовується для формування одного каналу. Ядра кожного фільтра генерують одну версію кожного каналу, а фільтр загалом створює один загальний вихідний канал (рисунок 7.10)

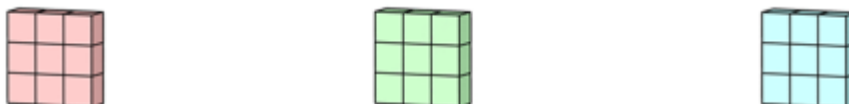


Рисунок 7.10 – Отримані версії кожного з каналів

Нарешті кожна вихідна версія має своє зміщення. Зміщення додається до вихідного каналу для створення кінцевого вихідного каналу.

Результат для будь-якої кількості фільтрів ідентичний: кожен фільтр обробляє вхід зі своїм відмінним від інших набором ядер і скалярним зміщенням по описаному вище процесу, створюючи один вихідний канал. Потім вони об'єднуються разом для отримання загального виходу, причому кількість вихідних каналів дорівнює кількості фільтрів. При цьому зазвичай застосовується нелінійність перед передачею входу іншому шару згортки, який повторює цей процес.

Striding

Часто буває, що при роботі з шаром згортки, потрібно отримати вихідні дані меншого розміру, ніж вхідні. Це зазвичай необхідно у згорткових нейронних мережах, де розмір просторових розмірів зменшується зі збільшенням кількості каналів. Один із способів досягнення цього – використання субдискритизуючих шарів (pooling layer), наприклад, приймати середнє/максимальне значення кожної гілки розміром 2 на 2, щоб зменшити всі просторові розміри вдвічі. Ще один спосіб досягти цього – використовувати stride (крок).

Ідея stride полягає в тому, щоб пропустити деякі області, над якими ковзає ядро. Крок 1 означає, що беруться прольоти через піксель, тобто за фактом кожен проліт є стандартною згорткою. Крок 2 означає, що прольоти

відбуваються через кожні два пікселі, пропускаючи всі інші прольоти в процесі і зменшуючи їх кількість приблизно в 2 рази, крок 3 означає пропуск 3 пікселів, скорочуючи кількість в 3 рази і т.д.

Більш сучасні мережі, такі як архітектури ResNet, повністю відмовляються від субдискритизуючих шарів у внутрішніх шарах на користь згорток, що чергуються, коли необхідно зменшити розмір на виході.

Таким чином, Stride (крок) – це кількість пікселів, зрушених по вхідній матриці. Коли крок дорівнює 1, ми переміщуємо фільтри на 1 піксель за раз. Коли крок дорівнює 2, ми переміщуємо фільтри на 2 пікселі за раз і таке інше. На рисунку 7.11 показано, що згортка буде працювати з кроком 2.

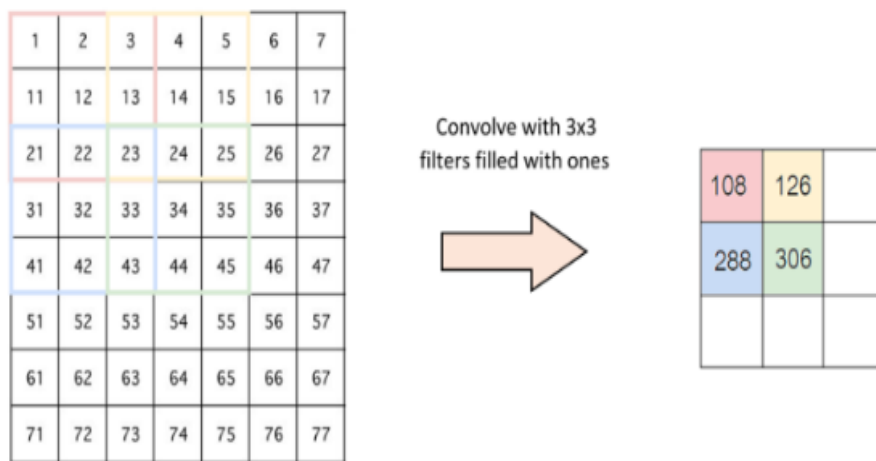


Рисунок 7.11 – Крок (stride) у 2 пікселі

Padding

Padding додає до країв підроблені (*fake*) пікселі (зазвичай нульового значення, тому до них застосовується термін "нульове доповнення" – "zero padding"). Таким чином, ядро при проходженні дозволяє непідробним пікселям виявлятися у центрі, а потім поширюється на підроблені пікселі за межами краю, створюючи вихідну матрицю того ж розміру, що і вхідна.

Отже, іноді фільтр ідеально не підходить для вхідного зображення. У нас є два варіанти:

- Додати до картинки нулі (заповнення нулями), щоб вона відповідала розмірам

- Відкинути частину зображення, де фільтр не міститься. Це називається допустимим заповненням, яке зберігає лише дійсну частину зображення.

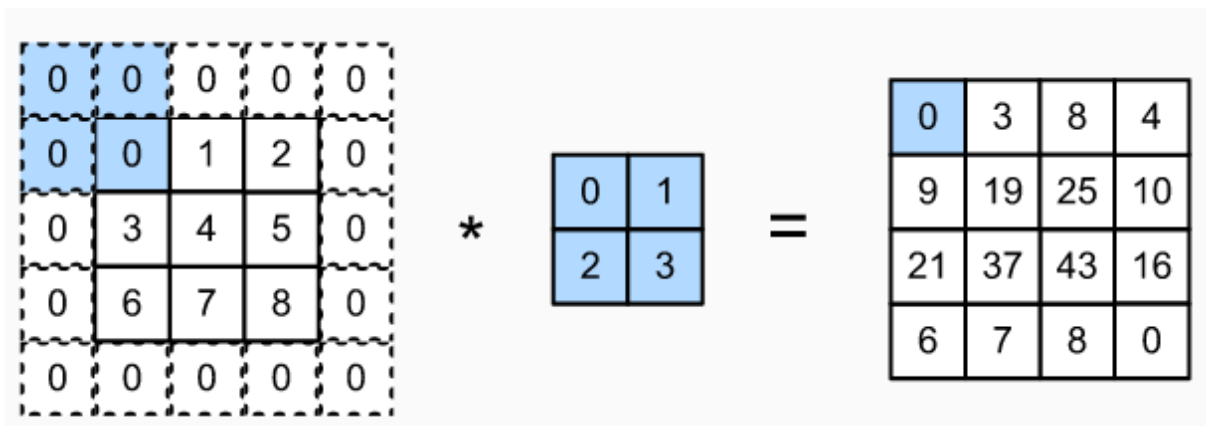


Рисунок 7.12 – Приклад згортки з padding = 1

Що таке padding same та valid (рисунок 7.13)?

Коли padding = "valid" – це значить, що вхідне зображення без padding (не додається фейкових полів). Це означає, що вікно фільтра завжди залишається всередині вхідного зображення. Цей тип padding називається valid, оскільки для цього padding розглядаються лише дійсні (valid) та оригінальні елементи вхідного зображення. Коли padding = "valid", може бути втрата інформації. Як правило, елементи праворуч і нижня частина зображення, ігноруються. Скільки елементів ігноруються, залежить від розміру ядра та кроку (stride).

Коли padding = "same", вхід half padded. Тип padding називається same, оскільки розмір виходу виходить такий же, як і розмір входу (коли крок = 1). Використання "same" гарантує, що фільтр застосовується до всіх елементів входу. Зазвичай padding встановлюється у значення "same" під час навчання моделі. Розмір виходу математично зручний для подальших обчислень.

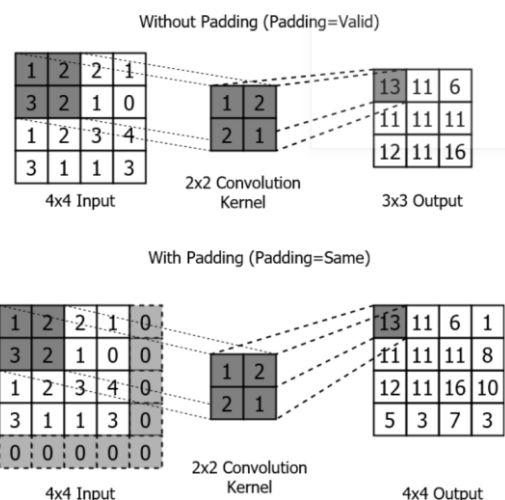


Рисунок 7.13 – Приклад SAME та VALID padding

(<https://livebook.manning.com/book/tensorflow-in-action/chapter-4/v-6/157>)

Pooling. Просторове об'єднання

Pooling зменшить кількість параметрів, коли зображення надто великі. Просторове об'єднання також називається субдискретизацією або знижувальною вибіркою, яка зменшує розмірність кожної картки ознак, але зберігає важливу інформацію. Просторовий pooling буває різних типів:

- Максимальне об'єднання (Max Pool, рисунок 7.14)
- Середній пул (Average Pool)
- Сума (Sum Pool)

Максимальний пул, наприклад, бере найбільший елемент із карти ознак. Average Pool бере середнє значення із карти ознак. Sum Pool – бере суму всіх елементів у карті ознак.

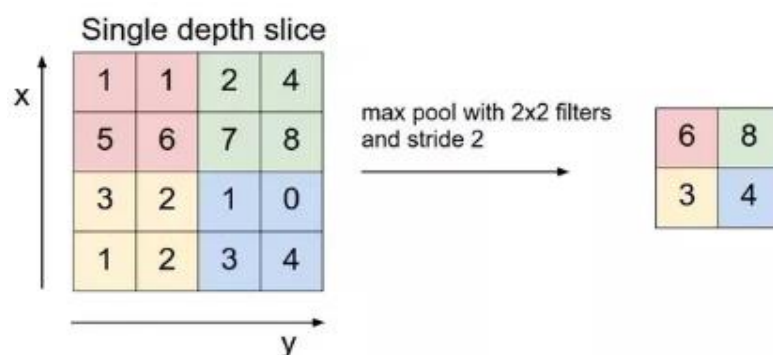


Рисунок 7.14 – Приклад Max Pooling

Параметри в згортковій нейронній мережі

Отже, згортка – це лінійне перетворення. Навіть з вже описаною механікою роботи згорткового шару, все ще складно пов'язати це з нейронною мережею прямого поширення (feed-forward network), і це все ще не пояснює, чому згортки масштабуються і працюють набагато краще із зображеннями.

Припустимо, що ми маємо вхід 4 на 4, і хочемо перетворити його у сітку 2 на 2. Якби ми використовували feed-forward network, ми переробили б вхід 4 на 4 у вектор довжиною 16 і передали його через повнозв'язковий шар з 16 входами і 4 виходами. Можна було б візуалізувати вагову матрицю W для шару на кшталт:

$$\begin{pmatrix} w_{1,1} & \cdots & w_{1,16} \\ \vdots & \ddots & \vdots \\ w_{4,1} & \cdots & w_{4,16} \end{pmatrix}$$

І хоча згорткові операції з ядрами можуть спочатку здатися трохи дивними, це, як і раніше, лінійні перетворення з еквівалентною матрицею переходу.

Якщо ми використовували ядро K розміру 3 видозміненим входом розміру 4 на 4, щоб отримати вихід 2 на 2, еквівалентна матриця переходу буде виглядати як показано на рисунку 7.15.

$$\begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 & 0 & 0 & 0 & 0 \\ 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 \\ 0 & 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}$$

Рисунок 7.15 – Еквівалентна матриця переходу

Примітка: коли наведена матриця є еквівалентною матрицею переходу, фактична операція зазвичай реалізується як зовсім інше матричне множення.

Згортка, в цілому, все ще є лінійним перетворенням, але в той же час вона також є зовсім іншим видом перетворення. Для матриці з 64 елементами існує лише 9 параметрів, які повторно використовуються кілька

разів. Кожен вихідний вузол отримує лише певну кількість входів (ті, що знаходяться всередині ядра). Немає жодної взаємодії з іншими входами, тому що вага для них дорівнює 0.

Корисно представляти згорткові операції як *hard prior* для вагових матриць. У даному контексті, під *prior* маються на увазі заздалегідь визначенні параметри мережі. Наприклад, коли ви використовуєте попередньо оброблену модель для класифікації зображень, ви використовуєте попередні параметри мережі як *prior*, як екстрактор образів для останнього повнозв'язного шару.

Transfer learning (метод адаптації однієї моделі для використання для задачі із схожої предметної області, шляхом налаштування тільки певної частини параметрів) ефективніший на порядок у порівнянні з випадковою ініціалізацією, тому що вам тільки потрібно оптимізувати параметри кінцевого повністю пов'язаного шару, а це означає, що ви можете мати фантастичну продуктивність лише з кількома десятками зображень у класі.

Вам не потрібно оптимізувати всі 64 параметри, тому більшість із них встановлена на нуль (і вони залишаються такими), а решту ми перетворимо на загальні параметри і в результаті отримаємо лише 9 параметрів для оптимізації. Ця ефективність має значення, тому що коли ви переходите від 784 входів MNIST до реальних зображень 224 на 224 на 3, це більше 150 000 входів. Шар, який намагається вдвічі зменшити вхід до 75 000 вхідних значень, як і раніше, вимагатиме понад 10 мільярдів параметрів. Для порівняння архітектура ResNet-50 має близько 25 мільйонів параметрів.

Таким чином, фіксування деяких параметрів до нуля і їх зв'язування підвищує ефективність, але на відміну від випадку з *transfer learning*, де ми знаємо, що *prior* працює грамотно, тому що він добре працює з великим загальним набором зображень, звідки ми знаємо, що це буде працювати хоч скільки добре в нашому випадку?

Відповідь полягає у комбінаціях образів, що вивчаються параметрами за рахунок *prior*.

Локальні особливості

Отже:

- Ядра поєднують пікселі лише з невеликої локальної області для формування виходу. Тобто вихідні ознаки бачать лише вхідні ознаки із невеликої локальної області;
- Ядро застосовується глобально по всьому зображенню для створення матриці вихідних значень.

Таким чином, з `backpropagation` (метод зворотного поширення помилки), що йде у всіх напрямках від вузлів класифікації мережі, ядра мають цікаве завдання вивчення ваги для створення ознак тільки з локального набору входів. Крім того, оскільки саме ядро застосовується по всьому зображенню, ознаки, які вивчає ядро, повинні бути достатньо спільними, щоб надходити з будь-якої частини зображення.

Якщо це були якісь інші дані, наприклад, дані про установки додатків за категоріями, то це стало б катастрофою, тому що кількість стовпців установки додатків і типів додатків поруч один з одним не означає, що вони мають «локальні загальні ознаки», загальні з датами встановлення програм та часом використання. Звичайно, у кількох можуть бути основні ознаки вищого рівня, які можуть бути знайдені, але це не дає нам жодних підстав вважати, що параметри для перших двох такі самі, як параметри для останніх двох. Ці декілька могли бути в будь-якому (послідовному) порядку і, як і раніше, залишатися підходящими!

Пікселі, однак, завжди відображаються в послідовному порядку, а сусідні пікселі впливають на піксель поруч, наприклад, якщо всі сусідні пікселі червоні, досить ймовірно, що піксель також червоний. Якщо є відхилення, це цікава аномалія, яка може бути перетворена на ознаку і все це можна виявити при порівнянні пікселя зі своїми сусідами, з іншими пікселями у своїй місцевості [35-50].

Ця ідея – те, на чому були засновані більш ранні методи отримання ознак комп'ютерним зором. Наприклад, для виявлення граней можна

використовувати фільтр виявлення граней *Sobel* – це ядро з фіксованими параметрами, що діє так само, як стандартна одноканальна згортка (рисунок 7.16)

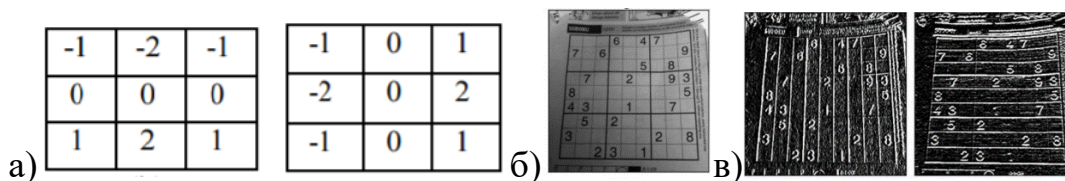


Рисунок 7.16 – Приклад фільтру Sobel для детектування граней (а), оригінальне зображення (б), отримане зображення після обох фільтрів (в)

Приклад застосування ядра, що детектує межі

Для сітки, що не містить граней (наприклад, неба на задньому фоні), більшість пікселів мають однакове значення, тому загальний вивід ядра в цій точці дорівнює 0. Для сітки з вертикальними гранями існує різниця між пікселями зліва і праворуч від грані, і ядро обчислює цю ненульову різницю, знаходячи ребра. Ядро за раз працює лише з сітками 3 на 3, виявляючи аномалії в певних місцях, але застосування до всього зображення достатньо для виявлення певної ознаки в будь-якому місці зображення!

Але чи корисні ядра можуть бути вивчені? Для ранніх шарів, що працюють з необробленими пікселями, ми могли б очікувати на детектори ознак низького рівня, таких як ребра, лінії і т.д.

Існує цілий напрямок досліджень глибокого навчання, орієнтований на те щоб зробити моделі нейронних мереж інтерпретованими. Один із найпотужніших інструментів для цього – візуалізація ознак за допомогою оптимізації. Ідея докорінно проста: оптимізуйте зображення (зазвичай ініціалізовано випадковим шумом), щоб активувати фільтр якомога сильніше. Такий спосіб інтуїтивно зрозумілий: якщо оптимізоване зображення повністю заповнене гранями, це переконливий доказ того, що фільтр активований і зайнятий пошуком. Використовуючи це, ми можемо зазирнути у вивчені фільтри, і результати будуть приголомшливими (рисунок 7.17, 7.18). Зверніть увагу, що хоча вони виявляють різні типи ребер, вони все ще є низькорівневими детекторами.

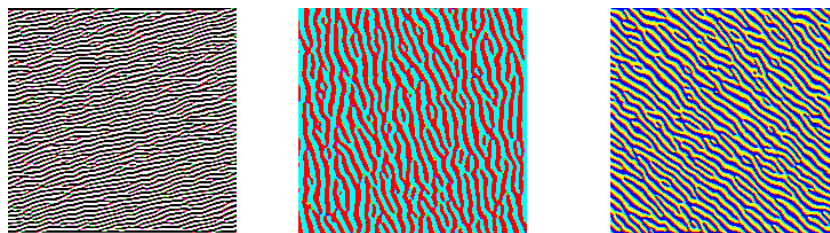


Рисунок 7.17 – Візуалізація ознак для 3 каналів після першого згорткового шару.

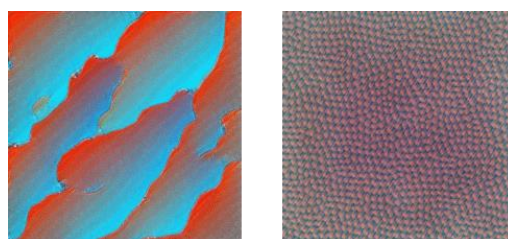


Рисунок 7.18 – Візуалізація після 2-ї та 3-ї згортки.

Важливо звернути увагу, що конвертовані зображення залишаються зображеннями. Вихід, який отримується від невеликої сітки пікселів у верхньому лівому кутку, буде теж розташований у верхньому лівому кутку. Таким чином, можна застосовувати один шар поверх іншого для отримання більш грубих ознак, які ми візуалізуємо.

Проте, хоч би як глибоко не заходили наші детектори ознак, без будь-яких подальших змін вони все одно працюватимуть на дуже маленьких ділянках зображення. Незалежно від того, наскільки глибокими є ваші детектори, ви не зможете виявити обличчя в сітці 3 на 3. І тут виникає ідея рецептивного поля (*receptive field*).

Рецептивні поля

Істотною особливістю архітектур згорткової нейронної мережі є те, що розміри вхідних даних стають все меншими і меншими від початку до кінця мережі, а кількість каналів стає більшою. Це, як згадувалося раніше, часто робиться за допомогою *strides* чи *pooling* шарів. Локальна область визначає, які вхідні дані попереднього рівня будуть на виході наступного.

Receptive field (RF, сприйнятливе поле) визначає, яку область вихідного входу отримує вихід [38-40].

У глибокому навчанні сприйнятливе поле визначається як розмір області в вході, який виробляє функцію (рисунок 7.19). В основному, це міра асоціації вихідної функції (будь-якого шару) до вхідної області (патч). Перш ніж ми продовжимо далі, давайте роз'яснимо одну важливу річ: Інсайт: Ідея сприйнятливих полів стосується місцевих операцій (тобто згортання (convolution), об'єднання (pooling)).

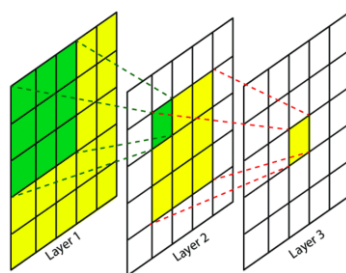


Рисунок 7.19 – Схематичне пояснення receptive field

Згортковий шар залежить лише від місцевого регіону (патч) входу. Ось чому ми ніколи не посилаємось на RF на повністю зв'язаних шарах, оскільки кожен блок має доступ до всієї області входу. З цією метою наша мета – дати зрозуміти цю концепцію, щоб зрозуміти та проаналізувати, як глибокі згорткові мережі працюють з місцевими операціями.

Ідея *strided convolution* полягає в тому, що ми обробляємо зображення тільки на фіксованій відстані один від одного та пропускаємо що посередині. З іншого погляду, ми залишаємо лише виходи на певній відстані друг від друга, видаляючи інші.

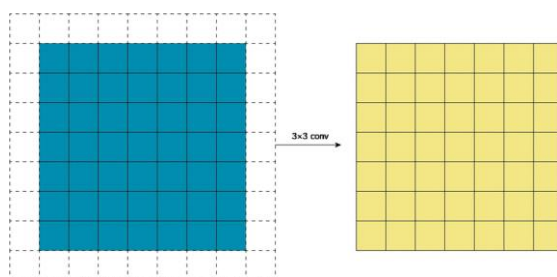


Рисунок 7.20 – Згортка 3 на 3, крок 2

Потім ми застосовуємо нелінійність до вихідних даних, потім накладаємо ще один новий шар згортки зверху. Тут усе стає цікавим. Навіть якби ми застосували ядро того ж розміру (3 на 3), що має одну й ту саму локальну область, до виходу *strided convolution*, ядро мало б більш ефективно *receptive field*.

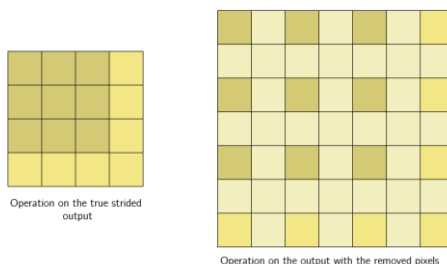


Рисунок 7.21 – Приклад *strided output*

Це пов'язано з тим, що вихід *strided* шару, як і раніше, являє собою одне і те ж зображення. Це не стільки обрізка, скільки зміна розміру, тільки тепер кожен окремий піксель на виході є представником більшої площі (інші пікселі якої були відкинуті) з того ж розташування вихідного входу. Тому коли ядро наступного шару застосовується до виходу, воно працює з пікселями, зібраними з більшої області.

Примітка: якщо ви знайомі з *розширеними згортками*, зверніть увагу, що вищезгадане нею не є. Обидва є методами збільшення *receptive field*, але *розширені згортки* є одним шаром, тоді як у нас все відбувається на регулярній згортці спільно з покроковою згорткою і нелінійністю між ними.

Таке розширення сприйнятливої області дозволяє шарам згортки поєднувати ознаки низького рівня (лінії, ребра) з ознаками вищого рівня (криві, текстури).

Після шарів *pooling/striding* мережа продовжує створювати детектори ще більш високорівневих ознак (частин, шаблонів).

Якщо ми повернемося до згорткової мережі (наприклад на рисунку 7.1), то повторне зменшення розміру зображення, скажемо, до 5-го блоку згорток дає розміри введення всього 7 на 7 порівняно з входами 224 на 224.

У цей момент кожен окремий піксель є величезною сіткою розміром 32 на 32 пікселя.

Якщо на більш ранніх шарах активація виявляла грані, то тут активація на сітці 7 на 7 потрібна для виявлення більш складних образів, наприклад, птахів.

Мережа в цілому розвивається з невеликою кількості фільтрів (64 у випадку GoogLeNet), виявляючи функції низького рівня, до дуже великої кількості фільтрів (1024 в кінцевій згортці), кожен з яких шукає надзвичайно специфічні ознаки високого рівня. І далі застосовується остаточний шар – *pooling layer*, який згортає кожен піксель 7 на 7 в один піксель, кожен канал є детектором ознак із *receptive field*, еквівалентним усьому зображенню.

У порівнянні з тим, що зробила б стандартна *feedforward* мережа, висновок тут не вражає. Стандартна *feedforward* мережа створила б абстрактні вектори ознак з комбінації всіх пікселів у зображенні, вимагаючи обсягів даних, що важко навчаються.

Згорткова нейронна мережа з накладеними на неї *priors* починає навчання з вивчення детекторів ознак низького рівня, і коли шар за шаром її *receptive field* стають все більше, вчиться комбінувати ці низькорівневі ознаки в ознаки вищого рівня; не абстрактне поєднання кожного пікселя, а сильна візуальна ієрархія.

Виявляючи низькорівневі ознаки і використовуючи їх для виявлення ознак вищого рівня в міру покращення своєї візуальної ієрархії, вона в кінцевому підсумку може виявляти цілі візуальні концепції, такі як особи, птахи, дерева і т.д., саме це робить їх такими потужними та ефективними для зображень.

Нелінійність (ReLU)

Як ми вже знаємо з минулих розділів ReLU розшифровується як Rectified Linear Unit і означає нелінійну роботу. Результатом його використання буде $f(x) = \max(0, x)$ (рисунок 7.22).

Чому ReLU важливий для згорткових мереж: мета ReLU – внести нелінійність до згорткової мережі (ConvNet).

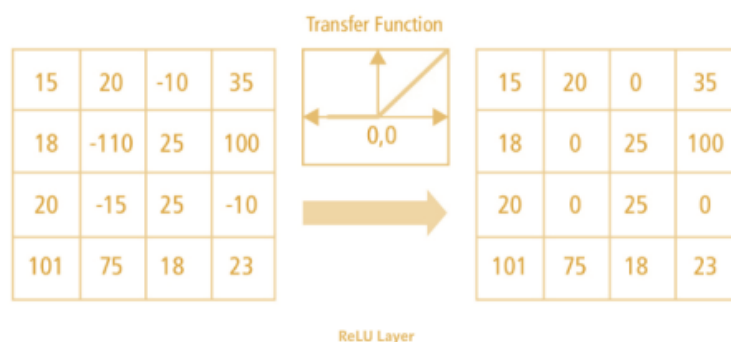


Рисунок 7.22 – Робота ReLU

Існують інші нелінійні функції, такі як $\tanh$ або sigmoid , які також можна використовувати замість ReLU. Однак, більшість фахівців з аналізу даних використовують ReLU, оскільки з точки зору продуктивності ReLU краще двох інших.

Повністю пов'язаний шар

Шар, який ми називаємо FC-шаром

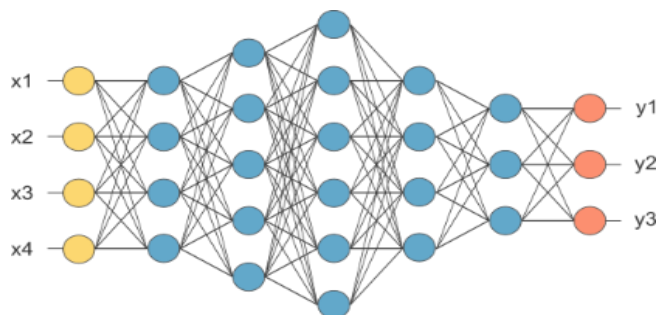


Рисунок 7.23 – Повнозв'язана мережа

На рисунку 7.23 матрицю карти ознак буде перетворено на вектор ($x_1, x_2, x_3, \dots$). З пов'язаними шарами ми об'єднали ці елементи разом, щоб створити модель.

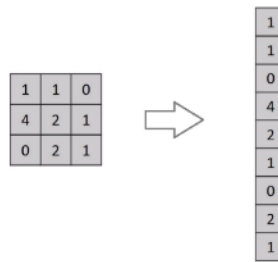


Рисунок 7.24 – Розгортання двовимірної матриці на вектор.

З рисунка 7.24 видно, що операція розгортання полягає в "склеюванні" рядків в єдиний – величезної довжини – числовий ряд. Це буде справді великий вектор, який потрібно буде ще й перетворити за допомогою багатошарової мережі з повними зв'язками!

Нагадаємо прости опис механізму, як працює повнозв'язковий шар: кожен елемент вектора множиться на вагу зв'язку, ці добутки далі підсумовуються між собою та з деяким зміщенням, після чого результат перетворюється за допомогою функції активації.

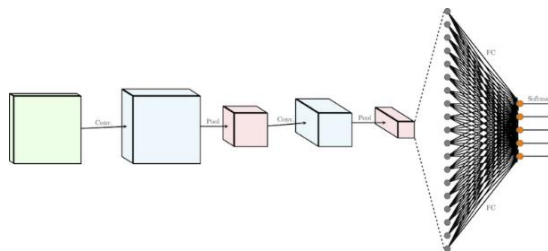


Рисунок 7.25 – Взаємодія згорткових шарів з багатошаровим перцептроном

Проблема adversarial attacks

Зі створенням візуальної ієрархії згорткової нейронної мережі цілком розумно припустити, що й системи бачення схожі на людську. І вони справді чудово справляються із зображеннями реального світу, але вони зазнають невдачі там, де їхня система не зовсім така як у людини. Найголовніша проблема: Adversarial Examples, приклади, які були спеціально змінені, щоб обдурити модель.

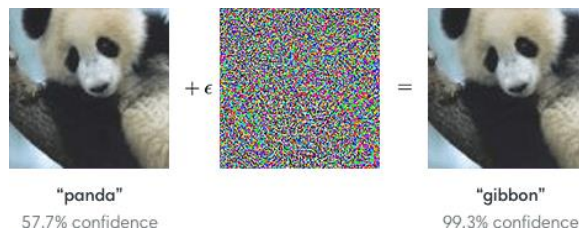


Рисунок 7.26 – Приклад adversarial attacks

Для людини очевидно, що на обох картинках на рисунку 7.26 зображено панду. Для комп'ютера – не зовсім.

Проблема в тому, що моделі сприйнятливі до зразків, які були трохи підроблені і не зможуть обдурити людину. Це відкриває двері для відмов у моделей, що небезпечно, наприклад, для самокерованих автомобілів та завдань охорони здоров'я.

Завдання захисту від таких відмов зараз є активною галуззю дослідження, предметом статей та конкурсів. Рішення безсумнівно покращить архітектуру згорткової нейронної мережі, зробить її надійнішою та безпечнішою.

Згорткові нейронні мережі дозволяють комп'ютерному зору працювати як із простими завданнями, так і зі складними продуктами та послугами, починаючи від розпізнавання осіб та закінчуючи покращенням медичних діагнозів. Вони можуть бути ключовим методом у комп'ютерному зорі у майбутньому. Одне можна сказати напевно: згорткові нейромережі – дивовижна основа багатьох сучасних інноваційних додатків, і, безумовно, заслуговують на глибоке вивчення [38-44].

Резюме, алгоритм роботи з CNN

- Надати вхідне зображення в згортковий шар
- Вибрати параметри, застосувати фільтри з кроками, за потреби додати відступ. Виконати згортку зображення та застосувати активацію ReLU до матриці.
- Виконати pooling, щоб зменшити розмір розмірності
- Додати стільки згорткових шарів, доки не отримаєте бажане

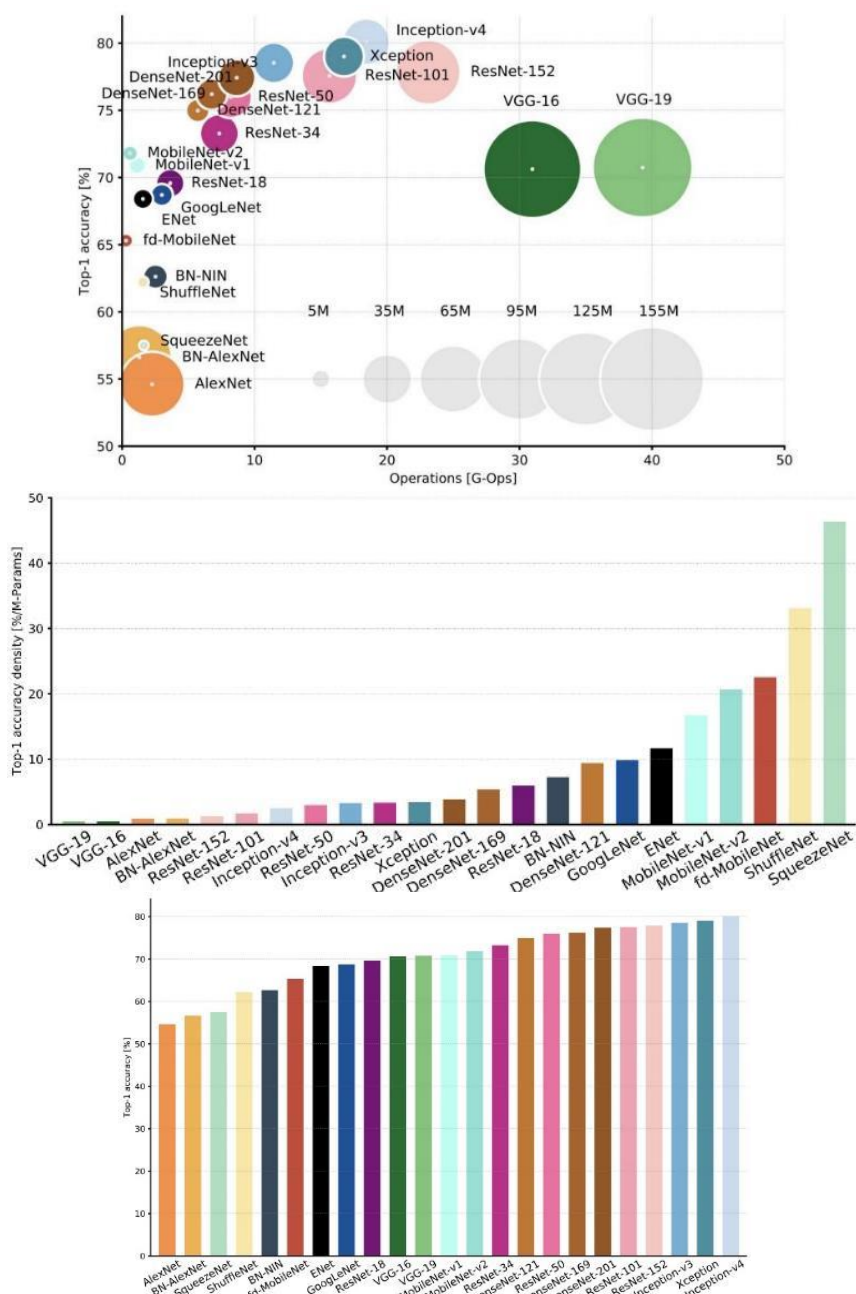
- Згладити вихід і подати на повністю пов'язаний шар (шар FC)
- Вивести клас за допомогою функції активації (логістична регресія з функціями вартості) та класифікувати зображення
- Розрахувати розмір вихідного зображення після операції згортки можна за формулою (вона є справедливою у разі різного кроку уздовж вертикального та горизонтального напрямку, підставляючи відповідні дані у вказаних напрямках):

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

де n_{in} – кількість вхідних ознак, n_{out} – кількість вихідних ознак, k – розмір ядра, p – розмір padding, s – розмір stride.

ТЕМА №8: АРХІТЕКТУРИ ЗГОРТКОВИХ НЕЙРОННИХ МЕРЕЖ.

Алгоритми глибоких нейромереж сьогодні набули великої популярності, яка багато в чому забезпечується продуманістю архітектур. Давайте розглянемо історію їхнього розвитку за останні кілька років (рисунок 8.1).



LeNet5

У 1994-му році була розроблена одна з перших згорткових нейромереж, яка започаткувала глибоке навчання [36]. Ця піонерська робота Яна Лекуна (Yann LeCun) після багатьох успішних ітерацій, починаючи з 1988-го, отримала назву LeNet5 (рисунок 8.2)

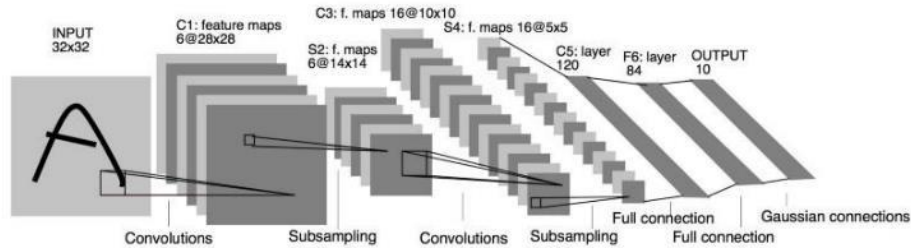


Рисунок 8.2 – Архітектура LeNet5

Архітектура LeNet5 стала фундаментальною для глибокого навчання, особливо з погляду розподілу властивостей зображення по всій картинці. Згортки з параметрами, що навчаються, дозволяли за допомогою декількох параметрів ефективно витягувати однакові властивості з різних місць. У ті роки ще не було відеокарт, здатних прискорити процес навчання, і навіть центральні процесори були повільними. Тому ключовою перевагою архітектури виявилася можливість зберігати параметри та результати обчислень, на відміну від використання кожного пікселя як окремі вхідні дані для великої багатошарової нейромережі. У LeNet5 у першому шарі пікселі не використовуються, тому що зображення сильно корельовані просторово, так що використання окремих пікселів у якості вхідних властивості не дозволить скористатися перевагами цих кореляцій.

Особливості LeNet5:

- Згорткова нейромережа, що використовує послідовність з трьох шарів: шари згортки (convolution), шари групування (pooling) і шари нелінійності (non-linearity). З моменту публікації роботи Лекуна це, мабуть, одна з головних особливостей глибокого навчання стосовно зображень.
- Використовує згортку для отримання просторових властивостей.
- Підвибір з використанням просторового усереднення карт.
- Нелінійність у вигляді гіперболічного тангенсу або сигмоїди.
- Фінальний класифікатор у вигляді багатошарової нейромережі (MLP).

- Розріджена матриця зв'язків між шарами дозволяє зменшити обсяг обчислень.

Ця нейромережа лягла в основу багатьох наступних архітектур і надихнула безліч дослідників.

Далі з 1998-го по 2010-й нейромережі перебували у стані інкубації. Більшість людей не помічали їхніх можливостей, хоча багато розробників поступово відточували алгоритми. Завдяки розквіту камер мобільних телефонів і здешевленню цифрових фотоапаратів становилося доступним все більше даних для навчання. Заодно зростали і обчислювальні можливості, процесори ставали потужнішими, а відеокарти перетворилися на основний обчислювальний інструмент. Всі ці процеси дозволяли розвиватися і нейромережам, хай і досить повільно. Зростав інтерес до завдань, які можна було вирішити за допомогою нейромереж, і нарешті ситуація стала очевидною.

Dan Ciresan Net

У 2010-му Дан Кіресан (Dan Claudiu Ciresan) та Йорген Шмідхубер (Jurgen Schmidhuber) опублікували один з перших описів реалізації GPU-нейромереж (<http://arxiv.org/abs/1003.0358>) [45]. Їхня робота містила пряму та зворотну реалізацію 9-шарової нейромережі на NVIDIA GTX 280 (<http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-280>).

AlexNet

У 2012-му Олексій Крижевський опублікував AlexNet (<https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>), поглиблену та розширену версію LeNet, яка з великим відривом перемогла у складному змаганні ImageNet (рисунок 8.3) [42].

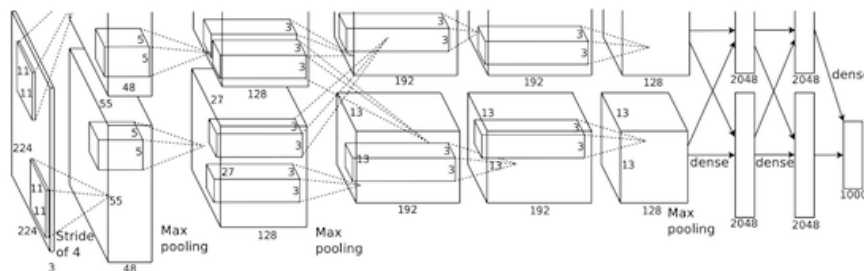


Рисунок 8.3 – Схематична архітектура AlexNet

В AlexNet результати обчислень LeNet масштабовані в набагато більшу нейромережу, яка здатна вивчити набагато складніші об'єкти та їх ієрархії. Особливості цього рішення:

- Використання блоків лінійної ректифікації (ReLU) як нелінійності.
- Використання методики відкидання (дропаут) для вибіркового ігнорування окремих нейронів під час навчання, що дозволяє уникнути перенавчання моделі.
- Використання *max pooling*, що дозволяє уникнути ефектів усереднення *average pooling*.
- Використання NVIDIA GTX 580 для прискорення навчання.

На той час кількість ядер у відеокартах сильно зростає, що дозволило приблизно в 10 разів скоротити час навчання, і в результаті стало можливо використовувати куди більші датасети і картинки.

Успіх AlexNet запустив невелику революцію, згорткові нейромережі перетворилися на робочого коня глибокого навчання – цей термін відтепер означав «великі нейромережі, здатні вирішувати корисні завдання».

Докладніше про AlexNet

AlexNet – згорткова нейронна мережа, яка вплинула на розвиток машинного навчання, особливо – на алгоритми комп'ютерного зору. Мережа з великим відривом виграла конкурс із розпізнавання зображень ImageNet LSVRC-2012 у 2012 році (з кількістю помилок 15.3% проти 26.2% у другого місця).

Архітектура AlexNet схожа на створену Yann LeCun мережу LeNet (рисунки 8.4). Однак у AlexNet більше фільтрів на шарі та вкладених згорткових шарів. Мережа включає згортки, максимальне об'єднання, дропаут, аугментацію даних, функції активації ReLU та стохастичний градієнтний спуск.

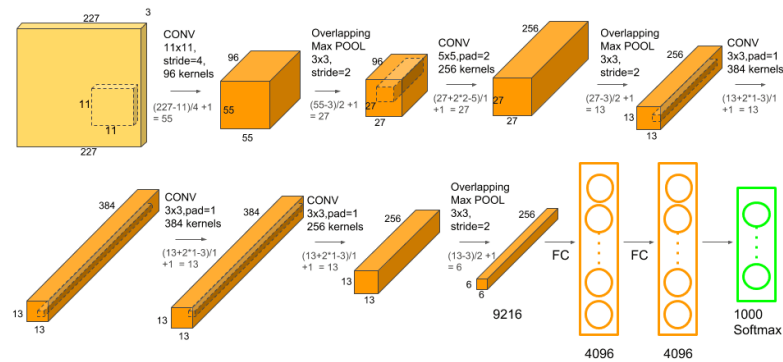


Рисунок 8.4 – Детальна архітектура AlexNet

Особливості AlexNet

1. Як функція активації використовується Relu замість арктангенсу для додавання моделі нелінійності. За рахунок цього при однаковій точності методу швидкість стає у 6 разів швидше.
2. Використання дропаута замість регуляризації вирішує проблему перенавчання. Однак час навчання подвоюється із показником дропауту 0.5.
3. Проводиться перекриття об'єднань (max pooling) для зменшення розміру мережі. За рахунок цього рівень помилок першого та п'ятого рівнів знижуються до 0.4% та 0.3%, відповідно.

Датасет ImageNet

ImageNet – це набір із 15 мільйонів позначених зображень із високою роздільною здатністю, розділених на 22 000 категорій. Зображення зібрані в інтернеті та позначені вручну за допомогою краудсорсингу Amazon's Mechanical Turk. Починаючи з 2010 року, проводиться щорічний конкурс ImageNet Large-Scale Visual Recognition Challenge (ILSVRC), що є частиною Pascal Visual Object Challenge. У челенджі використовується частина датасету ImageNet з 1000 зображень у кожній із 1000 категорій. Усього виходить 1.2 мільйона зображень для навчання, 50 000 зображень для перевірки та 150 000 – для тестування. ImageNet складається з зображень з різною роздільною здатністю. Тому для конкурсу їх масштабують до фіксованої роздільної здатності 256×256 . Якщо спочатку зображення було прямокутним, його обрізають до квадрата в центрі зображення.

Архітектура

Архітектура мережі наведена на рисунках 8.3 та 8.4. AlexNet містить вісім шарів із ваговими коефіцієнтами. Перші п'ять із них згорткові, інші три – повнозв'язні. Вихідні дані пропускаються через функцію втрат softmax, що формує розподіл 1000 міток класів. Мережа максимізує багатолінійну логістичну регресію, що еквівалентно максимізації середнього з усіх навчальних випадків логарифму ймовірності правильного маркування за розподілом очікування. Ядра другого, четвертого і п'ятого згорткових шарів пов'язані тільки з тими картами ядра в попередньому шарі, які знаходяться на тому самому графічному процесорі. Ядра третього згорткового шару пов'язані з усіма картами ядер другого шару. Нейрони повнозв'язних шарів пов'язані з усіма нейронами попереднього шару.

Таким чином, AlexNet містить 5 згорткових шарів та 3 повнозв'язних шари. Relu застосовується після кожного згорткового та повнозв'язного шару. Дропаут застосовується перед першим та другим повнозв'язними шарами. Мережа містить 62.3 мільйона параметрів та витрачає 1.1 мільярда обчислень при прямому проході. Згорткові шари, на які припадає 6% всіх параметрів, виробляють 95% обчислень.

Навчання

AlexNet проходить 90 епох. Навчання займає 6 днів одночасно на двох графічних процесорах Nvidia Geforce GTX 580, що є причиною того, що мережа поділена на дві частини. Використовується стохастичний градієнтний спуск зі швидкістю навчання 0.01, імпульсом 0.9 і розпадом вагових коефіцієнтів 0.0005. Швидкість навчання ділиться на 10 після насичення точності та знижується у 3 рази протягом навчання. Схема оновлення вагових коефіцієнтів w має вигляд:

$$v_{i+1} = 0.9v_i - 0.0005\epsilon w_i - \epsilon \left\langle \frac{\partial L}{\partial w} \middle| w_i \right\rangle_{D_i}, \quad w_{i+1} = w_i + v_{i+1}$$

де i – номер ітерації, v – змінна імпульсу, а ϵ – швидкість навчання. У ході всього етапу навчання швидкість навчання вибиралася рівною для всіх шарів і коректувалася вручну. Наступна евристика полягала в тому, щоб розділити швидкість навчання на 10, коли кількість помилок під час перевірки переставала зменшуватися.

Приклади використання та реалізація

Результати показують, що велика, глибока згорткова нейронна мережа здатна досягати рекордних результатів на дуже складних датасетах, використовуючи лише навчання з учителем. Через рік після публікації

AlexNet всі учасники конкурсу ImageNet почали використовувати згорткові нейронні мережі для вирішення завдання класифікації. AlexNet була першою реалізацією згорткових нейронних мереж і відкрила нову еру досліджень. Наразі реалізувати AlexNet стало простіше за допомогою бібліотек глибокого навчання: PyTorch, TensorFlow, Keras.

Результат

Мережа досягає наступного рівня помилок першого та п'ятого рівнів: 37.5% та 17.0%, відповідно. Найкраща продуктивність, досягнута в ході конкурсу ILSVRC-2010, становила 47,1% та 28,2% при використанні підходу, в якому усереднюються передбачення, отримані шістьма моделями з розрідженим кодуванням, навчених на різних векторах властивостей. З того часу досягнуто результатів: 45.7% і 25.7% при використанні підходу, в якому усереднюються передбачення двох класифікаторів, які навчаються на векторах Фішера.

Overfeat

У грудні 2013-го лабораторія NYU Яна Лекуна опублікувала опис Overfeat (<http://arxiv.org/abs/1312.6229>), різновиду AlexNet [46]. Також у статті описувалися bounding boxes, що навчаються, і згодом вийшло багато інших робіт з цієї тематики.

VGG

У розроблених в Оксфорді VGG-мережах у кожному згортковому шарі вперше застосували фільтри 3 на 3, та ще й об'єднали ці шари в послідовності згорток [47].

Це суперечить закладеним у LeNet принципам, згідно з якими великі згортки використовувалися для отримання однакових властивостей зображення. Замість фільтрів 9 на 9 і 11 на 11, що застосовуються в AlexNet, стали застосовувати набагато дрібніші фільтри, небезпечно близькі до згорток 1 на 1, яких намагалися уникнути автори LeNet, принаймні в перших шарах мережі. Але великою перевагою VGG стала знахідка, що кілька згорток 3 на 3, об'єднаних у послідовність, можуть емулювати великі рецептивні поля, наприклад, 5 на 5 або 7 на 7. Ці ідеї пізніше будуть використані в архітектурах Inception та ResNet.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A, A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

Рисунок 8.5 – Схематичне представлення архітектури VGG

Мережі VGG для представлення складних властивостей використовують численні шари згортки 3 на 3. Зверніть увагу на блоки 3, 4 і 5 у VGG-E (рисунок 8.5): для отримання більш складних властивостей та їх комбінування застосовуються послідовності 256 на 256 та 512 на 512 фільтрів 3 на 3. Це рівносильно великим згортковим класифікаторам 512 на 512 з трьома шарами! Це дає нам безліч параметрів і чудові здібності до навчання. Але навчати такі мережі було складно, доводилося розбивати їх на дрібніші, додаючи шари один за одним. Причина полягала у відсутності ефективних способів регуляризації моделей або якихось методів обмеження великого простору пошуку, якому сприяє безліч параметрів.

VGG у багатьох шарах використовують велику кількість властивостей, тому навчання вимагало великих обчислювальних витрат. Зменшити навантаження можна зменшивши кількість властивостей, як це зроблено в bottleneck-шарах архітектури Inception.

Докладніше про VGG.

Детальна архітектура VGG16 представлена на рисунку 8.6.

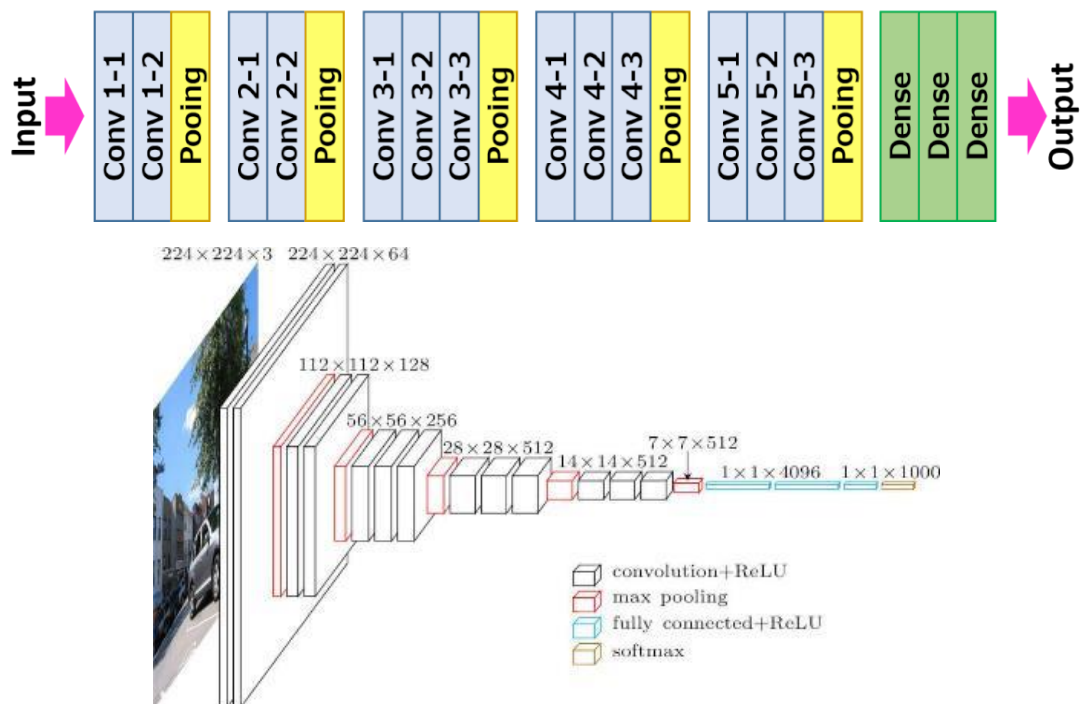


Рисунок 8.6 – Детальна архітектура VGG16

На вхід шару *conv1* подаються RGB зображення розміру 224 на 224. Далі зображення проходять через стек згорткових шарів, у яких використовуються фільтри з дуже маленьким рецептивним полем розміру 3 на 3 (який є найменшим розміром для отримання уявлення про те, де знаходиться право/ліво, верх/низ, центр).

В одній із конфігурацій використовується згортковий фільтр розміру 1 на 1, який може бути представлений як лінійна трансформація вхідних каналів (з наступною нелінійністю). Згортковий крок фіксується на значенні 1 піксель. Просторове доповнення (*padding*) входу згорткового шару вибирається таким чином, щоб просторова роздільна здатність зберігалася після згортки, тобто доповнення дорівнює 1 для 3 на 3 згорткових шарів. Просторовий пулінг здійснюється за допомогою п'яти *max-pooling* шарів, які йдуть за одним із згорткових шарів (не всі згорткові шари мають наступні *max-pooling*). Операція *max-pooling* виконується на вікні розміру 2 на 2 пікселів із кроком 2.

Після стека згорткових шарів (який має різну глибину в різних архітектурах) йдуть три повнозв'язкові шари: перші два мають по 4096 каналів, третій – 1000 каналів (оскільки у змаганні ILSVRC потрібно класифікувати об'єкти за 1000 категоріями), отже, класу відповідає один

канал. Останнім іде softmax шар. Конфігурація повнозв'язкових шарів та сама, що у всіх нейромережах.

Всі приховані шари забезпечені ReLU. Відзначимо також, що мережі (за винятком однієї) не містять шару нормалізації (Local Response Normalisation), оскільки нормалізація не покращує результату на датасеті ILSVRC, а веде до збільшення споживання пам'яті та часу виконання коду.

Конфігурація

Зміни згорткових мереж представлені рисунку 8.5. Кожна мережа відповідає своєму імені (A-E). Всі конфігурації мають загальну конструкцію, представлену в архітектурі, і відрізняються тільки глибиною: від 11 шарів з вагами в мережі A (8 згорткових та 3 повнозв'язкових шари) до 19 (16 згорткових та 3 повнозв'язкових шари). Ширина згорткових шарів (кількість каналів) відносно невелика: від 64 в першому шарі до 512 в останньому зі збільшенням кількості каналів у 2 рази після кожного max-pooling шару.

Реалізація

На жаль, мережа VGG має два серйозні недоліки:

1. Дуже повільна швидкість навчання.
2. Сама архітектура мережі важить занадто багато (з'являються проблеми з диском та пропускнуною спроможністю)

Через глибину та кількість пов'язаних вузлів, VGG16 важить більше 533 МБ. Це робить процес розгортання VGG стомлюючим завданням. Хоча VGG16 і використовується для вирішення багатьох проблем класифікації за допомогою нейронних мереж, менші архітектури кращі (SqueezeNet, GoogLeNet та інші). Незважаючи на недоліки, дана архітектура є чудовим будівельним блоком для навчання, оскільки її легко реалізувати.

Результати

VGG16 істотно перевершує у продуктивності минулі покоління моделей у змаганнях ILSVRC-2012 та ILSVRC-2013. Досягнутий VGG16 результат можна порівняти з переможцем змагання з класифікації (GoogLeNet з помилкою 6.7%) у 2014 році та значно випереджає результат Clarifai переможця ILSVRC-2013, який показав помилку 11.2% із зовнішніми тренувальними даними та 11.7% без них. Що стосується однієї мережі, архітектура VGG16 досягає найкращого результату (7.0% помилки на тесті), випереджаючи одну мережу GoogLeNet на 0.9%.

Таким чином, було показано, що глибина представлення позитивно впливає на точність класифікації, і state-of-the-art результат на датасеті змагання ImageNet може бути досягнутий за допомогою звичайної згорткової нейронної мережі з значно більшою глибиною.

Network-in-network

В основі архітектури Network-in-network (NiN) лежить проста ідея: використання згорток 1 на 1 для збільшення комбінаторності властивостей в згорткових шарах [48].

У NiN після кожної згортки застосовуються просторові MLP-шари, щоб краще скомбінувати властивості перед подачею до наступного шару. Може здатися, що використання згорток 1 на 1 суперечить вихідним принципам LeNet, але насправді це дозволяє комбінувати властивості краще, ніж просто набиваючи більше згорткових шарів. Цей підхід відрізняється від використання голих пікселів як вхідні дані для наступного шару. У цьому випадку згортки 1 на 1 застосовуються для просторового комбінування властивостей після згортки в рамках карт властивостей, так що можна використовувати набагато менше параметрів, які є спільними для всіх пікселів цих властивостей!

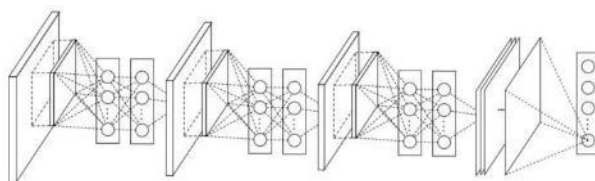


Рисунок 8.7 – Архітектура NiN мережі

MLP дозволяють сильно підвищити ефективність окремих згорткових шарів за допомогою їх комбінування у більш складні групи. Ця ідея пізніше була використана в інших архітектурах, таких як ResNet, Inception та їх варіантах.

GoogLeNet та Inception V1

Крістіан Жегеді (Christian Szegedy) з Google перейнявся зниженням обсягу обчислень у глибоких нейромережах, і в результаті створив GoogLeNet – першу архітектуру Inception (<https://arxiv.org/abs/1409.4842>) [49].

До осені 2014-го моделі глибокого навчання стали дуже корисними у категоризуванні вмісту зображень та кадрів з відео. Багато скептиків

визнали користь глибокого навчання і неймереж, а інтернет-гіганти, у тому числі Google, дуже зацікавилися розгортанням на своїх серверних потужностях ефективних і великих мереж.

Крістіан шукав шляхи зменшення обчислювального навантаження в неймережах, домагаючись високої продуктивності (наприклад, в ImageNet). Або зберігаючи обсяг обчислень, але при цьому підвищуючи продуктивність.

В результаті команда створила модуль Inception (рисунок 8.8)

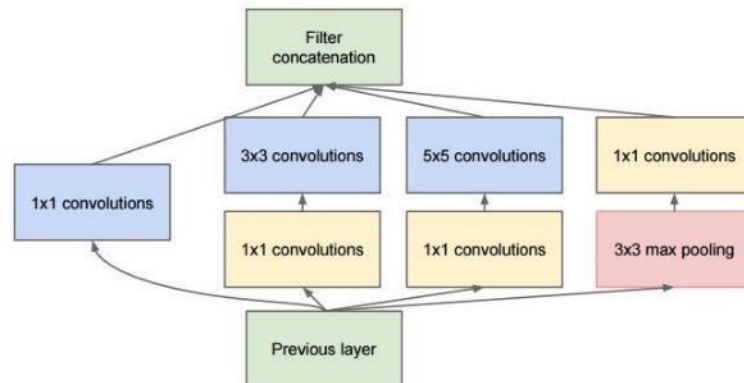


Рисунок 8.8 – Архітектура Inception

На перший погляд, це паралельна комбінація згорткових фільтрів 1 на 1, 3 на 3 та 5 на 5. Але особливість полягала у використанні згорткових блоків 1 на 1 (NiN) для зменшення кількості властивостей перед подачею в «дорогі» паралельні блоки. Зазвичай цю частину називають bottleneck.

У GoogLeNet як початковий шар застосовується stem без Inception-модулів, а також використовується average pooling і softmax-класифікатор, аналогічний NiN. Цей класифікатор виконує дуже мало операцій порівняно з AlexNet та VGG. Це також допомогло створити дуже ефективну архітектуру неймережі.

Bottleneck-шар

Цей шар зменшує кількість властивостей (а значить і операцій) у кожному шарі, тому швидкість отримання результату можна зберегти на високому рівні. Перш ніж передавати дані в «дорогі» згорткові модулі, кількість властивостей зменшується, скажімо, у 4 рази. Це сильно скорочує обсяг обчислень, що й забезпечило архітектурі популярність. Давайте розберемося. Нехай у нас 256 властивостей на вході та 256 на виході, і нехай Інсерпшн-шар виконує лише згортки 3 на 3. Отримуємо 256 на 256 на 3 на 3 згортку (589 000 операцій множення з накопиченням, тобто MAC-операцій).

Це може виходити за межі наших вимог до швидкості обчислень, припустимо, щоб шар оброблявся за 0.5 мілісекунд на Google Server. Тоді зменшимо кількість властивостей для згортання до 64 ($256/4$). У цьому випадку спочатку виконаємо згортку 1 на 1: для переходу з 256 до 64, потім ще 64 згортки у всіх Inception-гілках, а потім знову застосуємо згортку 1 на 1, для зворотного переходу з 64 до 256 властивостей. Кількість операцій:

- $256 \times 64 \times 1 \times 1 = 16\,000$
- $64 \times 64 \times 3 \times 3 = 36\,000$
- $64 \times 256 \times 1 \times 1 = 16\,000$

Усього близько 70 000, тобто знизили кількість операцій майже в 10 разів! Але при цьому ми не втратили узагальненості у цьому шарі. Bottleneck-шари продемонстрували чудову продуктивність на датасеті ImageNet, і стали застосовуватися в пізніших архітектурах, таких як ResNet. Причина їхнього успіху в тому, що вхідні властивості корельовані, а значить можна позбавитися надмірності, правильно комбінуючи властивості зі згортками 1 на 1. А після згортання з меншою кількістю властивостей можна на наступному шарі знову розгорнути їх у значну комбінацію.

Inception V2 та V3

Крістіан та його команда виявилися дуже ефективними дослідниками. У лютому 2015-го як друга версія Inception була представлена архітектура Batch-normalized Inception (<http://arxiv.org/abs/1502.03167>) [25]. Пакетна нормалізація (batch-normalization) обчислює середнє та середньоквадратичне відхилення всіх карт розподілу властивостей у вихідному шарі, та нормалізує їх відгуки з цими значеннями. Це відповідає «відбілюванню» даних, тобто відгуки всіх нейронних карт лежать в одному діапазоні і з нульовим середнім. Такий підхід полегшує навчання, оскільки наступний шар не повинен запам'ятовувати зміщення (offsets) вхідних даних і може займатися лише пошуком кращих комбінацій властивостей.

У грудні 2015-го вийшла нова версія модулів Inception та відповідної архітектури (<http://arxiv.org/abs/1512.00567>) [50]. В авторській статті краще пояснюється оригінальна архітектура GoogLeNet, там набагато докладніше розказано про ухвалені рішення. Основні ідеї:

- Максимізація потоку інформації у мережі за рахунок акуратного балансу між її глибиною і шириною. Перед кожним pooling збільшуються карти властивостей.

- Зі збільшенням глибини також систематично збільшується кількість властивостей або ширина шару.
- Ширина кожного шару збільшується для збільшення комбінації властивостей перед наступним шаром.
- В міру можливості використовуються тільки згортки 3 на 3. Враховуючи, що фільтри 5 на 5 та 7 на 7 можна декомпозувати за допомогою кількох згорток 3 на 3 (рисунок 8.9)

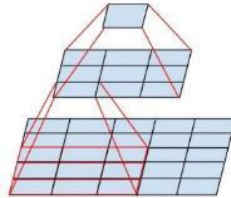


Рисунок 8.9 – Receptive field двох послідовних згорток 3 на 3

- Новий Inception-модуль виглядає як показано на рисунку 8.10.

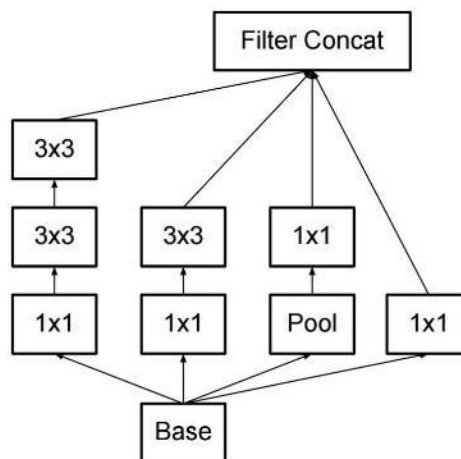


Рисунок 8.10 – Різновид Inception модуля V3

- Фільтри також можна декомпозувати за допомогою згладжених згорток у складніші модулі (рисунок 8.11)

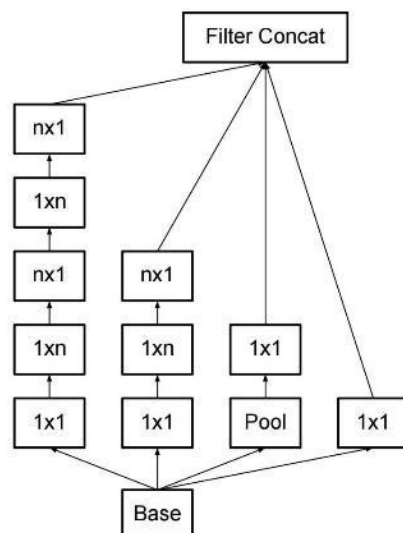


Рисунок 8.11 – Більш складний різновид Inception модуля V3

- Inception-модулі можуть за допомогою pooling-а в ході Inception-обчислень зменшувати розмір даних. Це аналогічно виконанню згортки зі страйдами паралельно з простим pooling-шаром (рисунок 8.12)

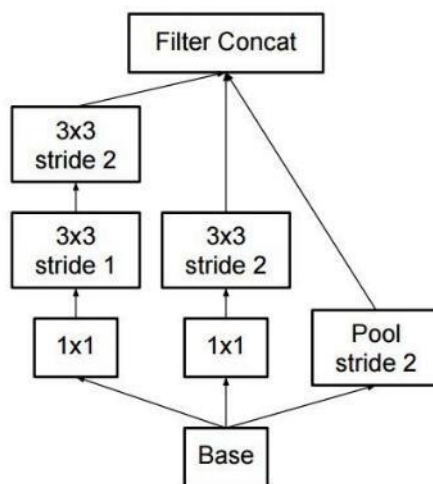


Рисунок 8.12 – Різновид Inception модуля V3 зі зміненими stride

Як фінальний класифікатор Inception використовує pooling-шар із softmax.

ResNet

У грудні 2015-го, приблизно в той же час, як було представлено архітектуру Inception v3, відбулася революція – опублікували ResNet. У ній закладені прості ідеї: подаємо вихідні дані двох успішних згорткових шарів ТА обходимо вхідні дані для наступного шару (рисунок 8.13) [51].

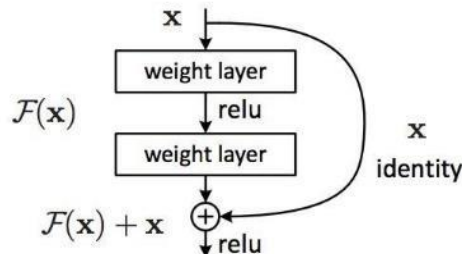


Рисунок 8.13 – Приклад residual connections

Такі ідеї вже пропонувалися, наприклад, тут <http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf> [52]. Але в даному випадку автори обходять ДВА шари та застосовують підхід у великих масштабах. Обхід одного шару не дає особливої вигоди, а обхід двох ключова знахідка. Це можна розглядати як маленький класифікатор, як *мережа в мережі*!

Також це був перший в історії приклад навчання мережі з кількох сотень, навіть тисячі верств. У багатошаровій ResNet застосували bottleneck-шар, аналогічний тому, що застосовується в Inception (рисунок 8.14)

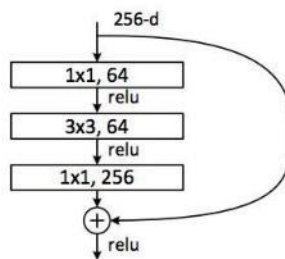


Рисунок 8.14 – Bottleneck шар у ResNet мережі

Цей шар зменшує кількість властивостей у кожному шарі, спочатку використовуючи згортку 1 на 1 з меншим виходом (зазвичай чверть від входу), потім йде шар 3 на 3, а потім знову згортка 1 на 1 в більшу кількість властивостей. Як і у випадку з Inception-модулями, це дозволяє заощаджувати обчислювальні ресурси, зберігаючи багатство комбінацій

властивостей. Порівняйте з більш складними і менш очевидними stem-ами Inception V3 і V4.

Як фінальний класифікатор у ResNet використовується pooling-шар із softmax. Додаткові відомості про архітектуру ResNet:

- Її можна розглядати як систему одночасно паралельних та послідовних модулів: у багатьох модулях inout-сигнал приходить паралельно, а вихідні сигнали кожного модуля з'єднуються послідовно.
- ResNet можна розглядати як кілька ансамблів паралельних чи послідовних модулів.
- ResNet, зазвичай, оперує блоками відносно невеликої глибини 20-30 шарів, що працюють паралельно, а не проганяючи послідовно по всій довжині мережі.
- Оскільки вихідний сигнал повертається назад і подається як вхідний, як це робиться в рекурентних мережах (про них мова буде йти у наступному розділі), ResNet можна вважати покращеною правдоподібною моделлю кори головного мозку.

Докладніше про ResNet

ResNet – це скорочена назва для Residual Network (дослівно – "залишкова мережа"), але що таке residual learning ("залишкове навчання")?

Глибокі згорткові нейронні мережі перевершили людський рівень класифікації зображень у 2015 році. Глибокі мережі отримують низько-, середньо-і високорівневі ознаки наскрізним багат шаровим способом, а збільшення кількості stacked layers може збагатити рівні ознак. Stacked layer має вирішальне значення (рисунок 8.15)

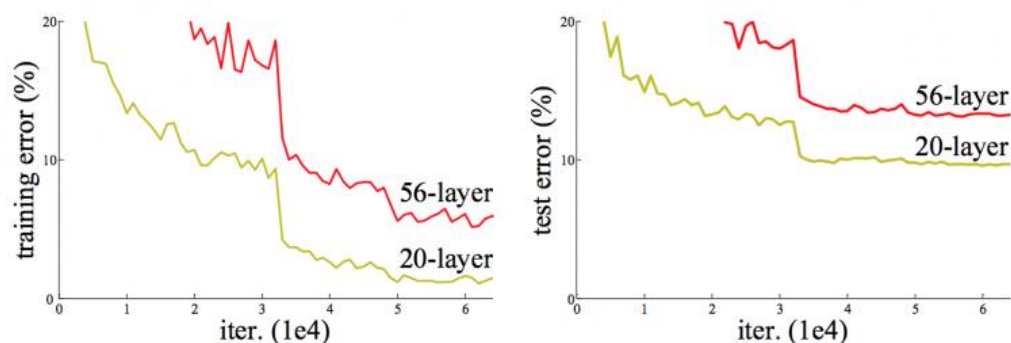


Рисунок 8.15 – Помилка навчання (ліворуч) та помилка тесту (праворуч) на CIFAR-10 з 20-рівневими та 56-шаровими «простими» мережами. Глибока мережа має велику помилку навчання і, отже, помилку тестування.

Коли глибша мережа починає згортатись, виникає проблема: зі збільшенням глибини мережі точність спочатку збільшується, а потім швидко погіршується. Зниження точності навчання показує, що не всі мережі легко оптимізувати.

З'єднання швидкого доступу

Саме щоб подолати цю проблему, Microsoft запровадила глибоку «залишкову» структуру навчання. Замість того, щоб сподіватися, що кожен кілька stacked layers безпосередньо відповідають бажаному основному уявленню, вони явно дозволяють цим шарам відповідати «залишковому». Формулювання $F(x)+x$ може бути реалізовано за допомогою нейронних мереж зі з'єднаннями для швидкого доступу (рисунк 8.13).

Отже, з'єднання швидкого доступу (shortcut connections) пропускають один або кілька шарів і виконують зіставлення ідентифікаторів. Їхні виходи додаються до виходів stacked layers. Використовуючи ResNet, можна вирішити безліч проблем, таких як:

- ResNet відносно легко оптимізувати: «прості» мережі (які просто складають шари) показують велику помилку навчання, коли глибина збільшується.
- ResNet дозволяє відносно легко збільшити точність завдяки збільшенню глибини, чого з іншими мережами досягти складніше.

Порівняння мереж для ImageNet надане на рисунку 8.16.

Проста мережа: Прості базові лінії (в центрі) здебільшого натхненні філософією мереж VGG (ліворуч на рисунку 8.16). Згорткові шари в основному мають фільтри 3 на 3 і дотримуються двох простих правил:

1. Для однієї і тієї ж вихідної карти об'єктів шари мають однакову кількість фільтрів;
2. Якщо розмір карти об'єктів зменшується вдвічі, кількість фільтрів подвоюється, щоб зберегти тимчасову складність кожного шару.

Зазначимо, що ResNet має менше фільтрів і складність ніж VGG.

ResNet: на основі описаної вище простої мережі додано швидке з'єднання (праворуч на рисунку 8.16), яке перетворює мережу на її залишкову версію. Ідентифікаційні швидкі з'єднання $F(x)+x$ можуть використовуватися безпосередньо, коли вхід та вихід мають однакові розмірності (швидкі з'єднання суцільної лінії на рисунку 8.16). Коли розмірності збільшуються (пунктирні лінії на рисунку 8.16), розглядається два варіанти:

2. Проекція швидкого з'єднання $F(x\{W\}+x)$ використовується для зіставлення розмірностей (виконано за допомогою згорток 1 на 1).

Для будь-якої з опцій, якщо швидкі з'єднання йдуть картами об'єктів двох розмірностей, вони виконуються з кроком 2.

Різновид архітектур ResNet

Кожен блок ResNet має два рівні глибини (використовується у невеликих мережах, таких як ResNet 18, 34) або 3 рівні (ResNet 50, 101, 152) (рисунок 8.17).

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Рисунок 8.17 – Різновид архітектур ResNet

50-шарова ResNet: кожен 3-шаровий блок замінюється в 34-шаровій мережі 3-шаровим вузьким місцем (bottleneck), в результаті виходить 50-шарова ResNet. Вони використовують варіант 2 для збільшення розмірностей. Ця модель має 3.8 мільярдів FLOPs.

ResNet зі 101 і 152 шарами: створюються використовуючи більше 3-шарових блоків. Навіть після збільшення глибини 152-шарова ResNet (11.3 мільярда FLOPs) має меншу складність, ніж мережі VGG-16/19 (15.3/19.6 мільярда FLOPs).

Реалізація

Розмір зображення змінюється за допомогою випадкової вибірки його короткої сторони [256, 480] для збільшення масштабу. Кадрування 224 на 224 вибирається випадковим чином із зображення або його горизонтального зміщення з відніманням середнього значення для кожного пікселя. Швидкість навчання стартує з 0.1 і ділиться на 10, коли зміна помилок виходить на плато, моделі навчаються до 60×10000 ітерацій. Автори також використовують зниження ваги 0.0001 та імпульс 0.9.

Результат

18-рівнева мережа – це просто підпростір у 34-рівневій мережі, і вона все ще працює краще. ResNet виграє зі значним відривом, якщо мережа глибша (рисунок 8.18).

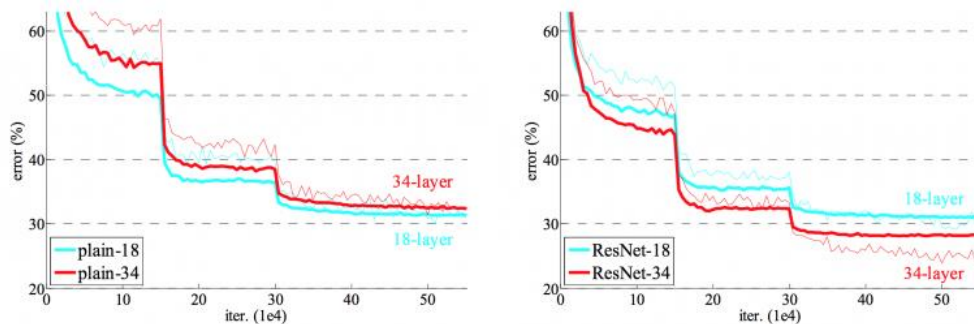


Рисунок 8.18 – Навчання у ImageNet. Тонкі криві позначають помилку навчання, а жирні криві позначають помилку валідації. Зліва: прості мережі з 18 та 34 шарів. Справа: ResNets з 18 та 34 шарів. На цьому графіку залишкові мережі не мають додаткових параметрів у порівнянні з їхніми звичайними аналогами.

Отже, мережа ResNet сходиться швидше, ніж її простий аналог. Рисунок 8.19 показує, що більш глибокі ResNet досягають кращих результатів навчання, порівняно з неглибокою мережею.

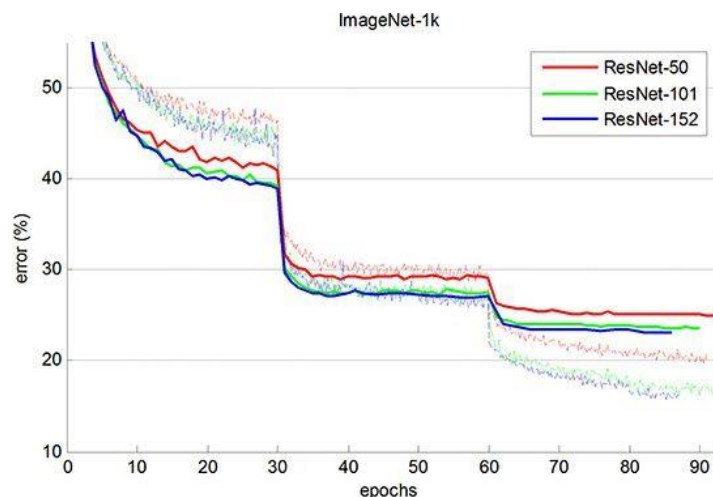


Рисунок 8.19 – результати навчання для більш глибоких ResNet

ResNet-34 досяг топ-5 помилок перевірки на 5.71% краще, ніж BN-start та VGG.

ResNet-152 досягає 4.49% у топ-5 помилок валідації. Комбінація з 6 моделей з різною глибиною досягає 3.57% у топ-5 помилок валідації.

Inception V4

Знову відзначилися Крістіан та його команда, які випустили нову версію Inception (<http://arxiv.org/abs/1602.07261>) [53]. Inception-модуль, що йде після stem, такий же, як у Inception V3 (рисунок 8.20)

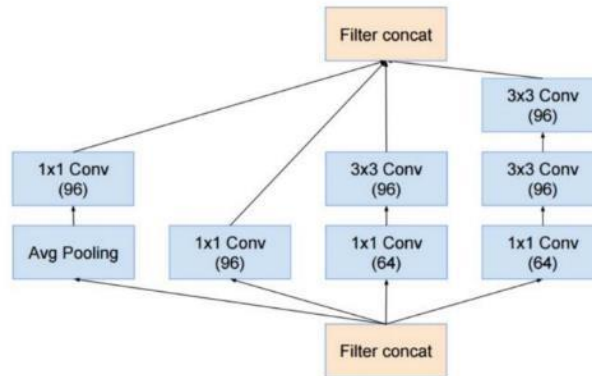


Рисунок 8.20 –Inception модуль у архітектурі V4, що йде після stem

При цьому Inception-модуль скомбінований із ResNet-модулем (рисунок 8.21)

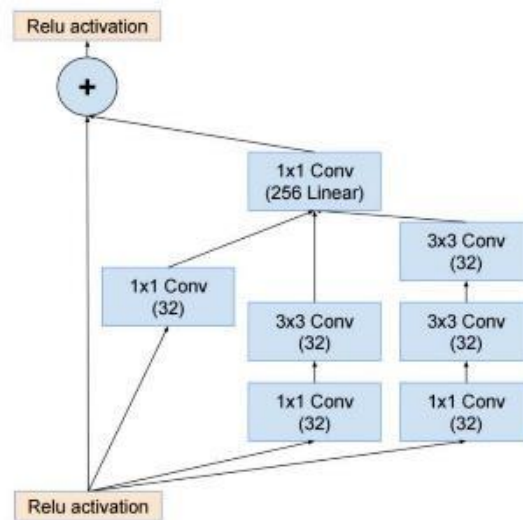


Рисунок 8.21 – Inception модуль із ResNet модулем

SqueezeNet

SqueezeNet опублікували у 2016 році (<http://arxiv.org/abs/1602.07360>) [54]. Це переробка на новий лад багатьох концепцій ResNet і Inception. Автори продемонстрували, що покращення архітектури дозволяє зменшити розміри мереж та кількість параметрів без складних алгоритмів стиснення.

ENet

Всі особливості попередніх архітектур скомбіновані в дуже ефективну та компактку мережу, що використовує зовсім мало параметрів та обчислювальних потужностей, але при цьому дає чудові результати. Архітектура отримала назву Enet (<https://arxiv.org/abs/1606.02147>) [55], її розробив Адам Пазке (Adam Paszke).

Це мережа на основі кодувальника та декодера. Кодувальник побудований за звичайною схемою CNN для категоризації, а декодер є мережею з підвищенням дискретизації (upsampling network), призначений для сегментування за допомогою поширення категорій назад у зображення вихідного розміру. Для сегментації зображень використовувалися лише нейромережі, жодних інших алгоритмів.

Enet під час її появи, мав найвищу питому точність порівняно з рештою нейромереж.

ENet створювалася з розрахунку, щоб від початку використовувати якнайменше ресурсів. В результаті кодувальник та декодер разом займають всього 0.7 Мб з точністю fp16. І при такому крихітному розмірі ENet за точністю сегментування не поступається або перевершує інші суто нейромережні рішення.

Xception

Xception (<https://arxiv.org/abs/1610.02357>) [56] привнесла до Inception-модуля більш просту та елегантну архітектуру, яка не менш ефективна, ніж ResNet та Inception V4 (рисунок 8.22).

Ця мережа сподобається будь-кому завдяки простоті та елегантності своєї архітектури.

Вона містить 36 етапів згортки, і в цьому схожа на ResNet-34. При цьому модель і код прості, як у ResNet, і набагато приємніші, ніж у Inception V4.

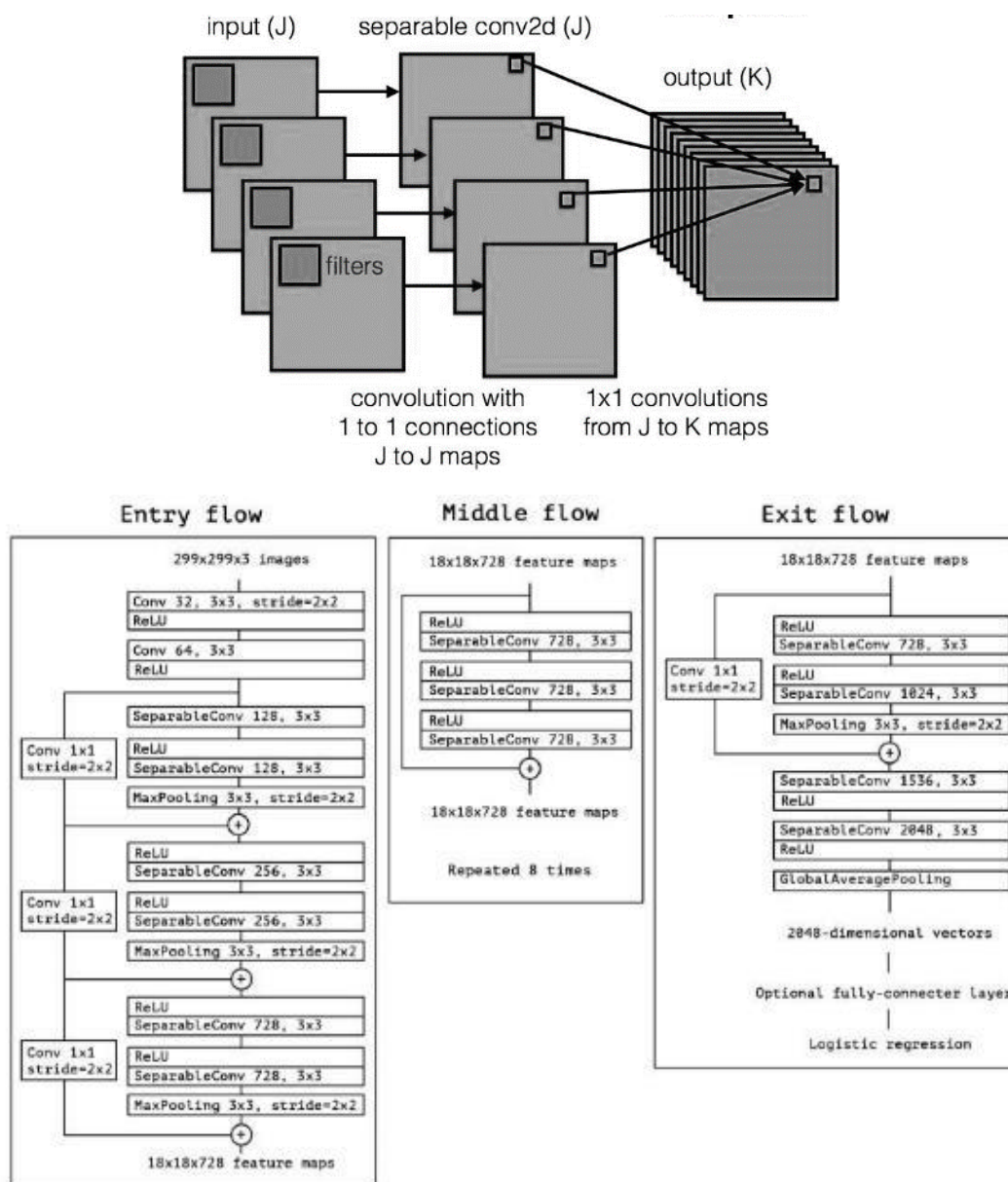


Рисунок 8.22 – Схематичне представлення Xception архітектури

MobileNets

Архітектура MobileNets (<https://arxiv.org/abs/1704.04861>) [57] вийшла у квітні 2017-го. Для зменшення кількості параметрів у ній використовуються згортки, що відокремлюються, такі ж, як у Xception. Ще в роботі стверджується, що автори змогли зменшити кількість параметрів: приблизно вдвічі у випадку з FaceNet. Повна архітектура моделі наведена на рисунку 8.23

Table 1. MobileNet Body Architecture		
Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32 \text{ dw}$	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64 \text{ dw}$	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128 \text{ dw}$	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128 \text{ dw}$	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256 \text{ dw}$	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256 \text{ dw}$	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
5×	Conv dw / s1	$3 \times 3 \times 512 \text{ dw}$
	Conv / s1	$1 \times 1 \times 512 \times 512$
	Conv dw / s2	$3 \times 3 \times 512 \text{ dw}$
	Conv / s1	$1 \times 1 \times 512 \times 1024$
	Conv dw / s2	$3 \times 3 \times 1024 \text{ dw}$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Рисунок 8.23 – Архітектура моделі MobileNets

FractalNet

Використовує рекурсивну архітектуру і є похідним або загальним варіантом ResNet [58].

U-Net: неймережа для сегментації зображень

U-Net вважається однією із стандартних архітектур CNN для завдань сегментації зображень, коли потрібно не тільки визначити клас зображення цілком, але й сегментувати його області за класом, тобто створити маску, яка розділятиме зображення на кілька класів. Архітектура складається з стягуючого шляху для захоплення контексту і симетричного шляху, що розширюється, що дозволяє здійснити точну локалізацію (рисунок 8.24) [59].

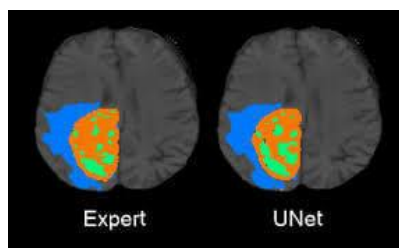


Рисунок 8.24 – Приклад сегментації мозку

Мережа навчається наскрізним способом на невеликій кількості зображень і перевершила попередній найкращий метод (згорткову мережу зі ковзним вікном) на змаганні ISBI з сегментації нейронних структур в електронно-мікроскопічних стеках. Використовуючи ту саму мережу, яка була навчена на зображеннях світлової мікроскопії (фазовий контраст та DIC), U-Net посіла перше місце у конкурсі ISBI 2015 року з трекінгу клітин у цих категоріях з великим відривом. Крім того, ця мережа працює швидко. Сегментація зображення 512 на 512 займає менше секунди на сучасному графічному процесорі.

Для U-Net характерно:

- досягнення високих результатів у різних реальних завданнях, особливо для біомедичних додатків;
- використання невеликої кількості даних для досягнення добрих результатів.

Архітектура U-Net (рисунок 8.25)

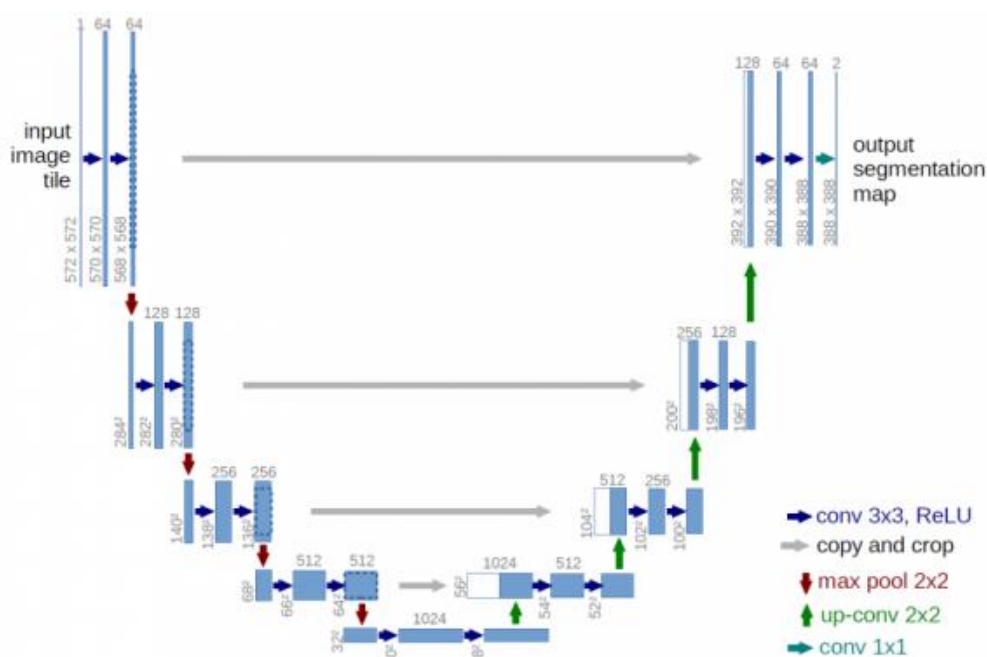


Рисунок 8.25 – Архітектура U-Net

Кожен синій квадрат відповідає багатоканальній карті властивостей. Кількість каналів наведено у верхній частині квадрата. Розмір x-y наведено в нижньому лівому краю квадрата. Білі квадрати є копією карти властивостей. Стрілки позначають різні операції.

Архітектура складається з звужувального шляху (ліворуч) і шляху, що розширюється (праворуч). *Шлях звуження* – це типова архітектура

згорткової нейронної мережі. Він складається з повторного застосування двох згорток 3 на 3, за якими слідує ReLU та операція максимального об'єднання (2 на 2 ступеня 2) для зниження роздільної здатності.

На кожному етапі знижувальної дискретизації канали властивостей подвоюються. Кожен крок у шляху, що розширюється, складається з операції, що підвищує дискретизації карти властивостей, за якою слідує:

- згортка 2 на 2, яка зменшує кількість каналів властивостей;
- об'єднання з відповідним чином обрізаною картою властивостей зі шляху, що стягується;
- дві 3 на 3 згортки, за якими слідує ReLU.

Обрізка необхідна через втрату граничних пікселів при кожній згортці (рисунок 8.26).

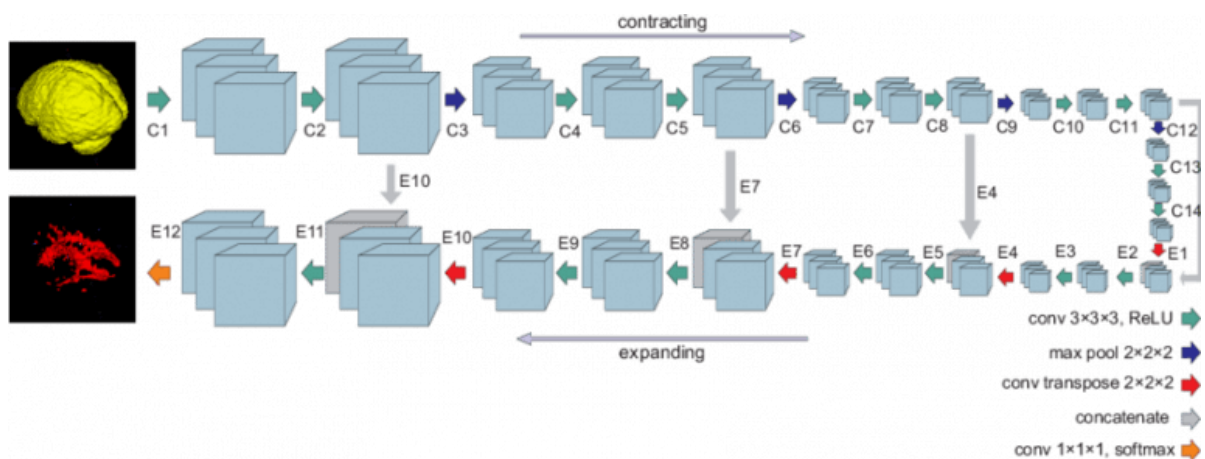


Рисунок 8.26 – Приклад схеми мережі U-net

На останньому шарі використовується згортка 1 на 1 для зіставлення кожного 64-компонентного вектора властивостей з бажаною кількістю класів. Усього мережа містить 23 згорткових шари.

Навчання U-Net

Мережа навчається методом стохастичного градієнтного спуску на основі вхідних зображень та відповідних карт сегментації. Через згортки вихідне зображення менше вхідного сигналу на постійну ширину границі. Функція softmax, що застосовується попіксельно, обчислює енергію по остаточній карті властивостей разом з функцією крос-ентропії. Границя поділу обчислюється з допомогою морфологічних операцій. Потім обчислюється карта вагових коефіцієнтів.

Приклади використання та реалізації

U-Net неодноразово використовувалася для сегментації зображень:

- класифікації нервів за допомогою ультразвуку
- сегментації кровоносних судин сітківки
- застосування невеликої U-net для розпізнавання автомобілів

Як видно з прикладів, ця мережа є універсальною і може використовуватися для будь-якої розумної задачі сегментації зображень. Висока точність досягається за умови належного навчання, відповідного датасету та правильного вибору часу навчання.

Результати

U-Net застосовувався у задачі сегментації клітин на зображеннях з оптичного мікроскопа. Це завдання сегментації є частиною конкурсів ISBI 2014 та 2015 років з трекінгу клітин.

Датасет PhC-U373 містить зображення клітини U373 Glioblastoma-astrocytoma на поліакриламідній підкладці, записані методом фазово-контрастної мікроскопії. Він містить 35 частково помічених зображень для навчання. У цьому випадку U-Net досягла середнього значення IOU (перетин по об'єднанню), що дорівнює 92%, що значно краще за попередній результат на той час: 83%.

Другий датасет DIC-HeLa містить фотографії клітин HeLa на плоскому склі, записані за допомогою мікроскопії диференціального інтерференційного контрасту (DIC) (рисунок 8.27). Він містить 20 частково помічених зображень для навчання. У цьому випадку U-Net досягла середнього значення IOU 77.5%, що значно краще ніж наступний алгоритм з його 46% (на час проведення змагання).

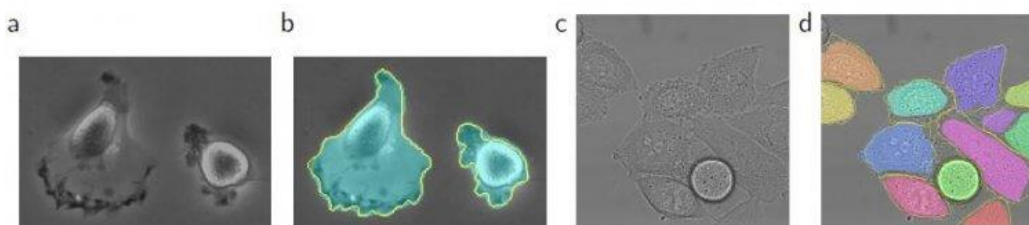


Рисунок 8.27 – Результати змагання ISBI з трекінгу клітин. (а) Частина вхідного зображення з датасету PhC-U373. (b) Результат сегментації (блакитна маска) та мітка (жовта межа). (с) Вхідне зображення датасету DIC-HeLa. (d) Результат сегментації (кольорові маски) та мітки (жовта рамка).

Архітектура U-net досягає видатної продуктивності і точності в різних додатках біомедичної сегментації. Метод вимагає лише кількох помічених зображень для тренування і має прийнятний час навчання: лише 10 годин на графічному процесорі NVidia Titan (6 ГБ).

NBDT: нейромережа на основі вирішальних дерев

Дослідники з UC Berkley запропонували інтерпретовану нейромережу, яка заснована на архітектурі вирішальних дерев та видає порівнянні з state-of-the-art передбачення [60]. Код проекту та передбачення моделі доступні у відкритому репозиторії на GitHub: <http://github.com/alvinwan/neuralbacked-decision-trees>.

Мотивація створення.

Глибоке навчання застосовується у сферах, які потребують обґрунтованості передбачень. Такими сферами є фінансове моделювання чи діагностування хвороб. Існуючі дослідження фокусуються на інтерпретованості навчених state-of-the-art нейромереж постфактум. До популяризації нейромереж вирішальні дерева були золотим стандартом як баланс між інтерпретованістю та точністю моделі.

Попередні спроби скомбінувати вирішальні дерева та нейромережі результували в моделі, які:

1. Менш точні, ніж сучасні нейромережі, навіть на порівняно невеликих датасетах (MNIST);
2. Потребують значних змін у архітектурах

Neural-Backed Decision Trees (NBDTs) видають state-of-the-art результати. При цьому вони не вимагають змін в архітектурі нейромережі. NBDTs по точності відрізняються від базової нейромережі не більше ніж на 1% на датасетах CIFAR10, CIFAR100 і TinyImageNet. На ImageNet запропонована архітектура по точності відстає, скажемо, від EfficientNet на 2%.

Як вона працює.

Процес навчання та інференсу (inference) Neural-Backed Decision Tree складається з чотирьох етапів (рисунок 8.28):

1. Спершу вибудовується ієрархія для вирішального дерева, яку називають Induced Hierarchy;

2. Ця ієрархія використовує спеціальну функцію втрати Tree Supervision Loss;
3. На інференсі дані спочатку проходять через базовий блок нейромережі (backbone), що йде до фінального повнозв'язного шару;
4. Фінальний повнозв'язковий шар проганяється як послідовність вирішальних правил, які називають Embedded Decision Rules

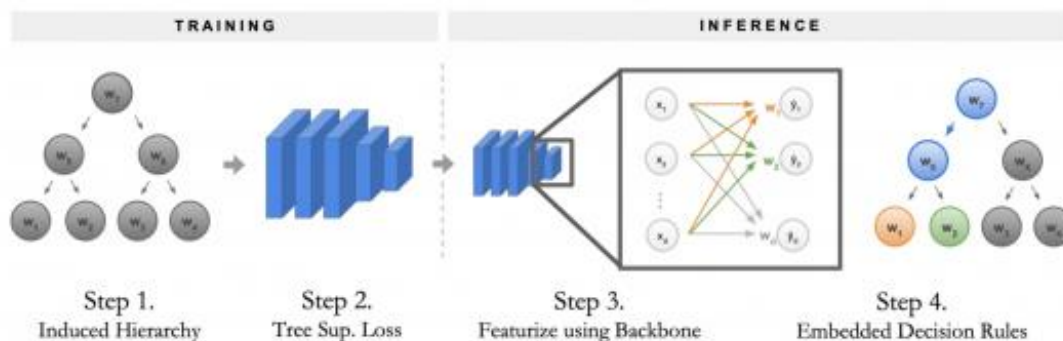


Рисунок 8.28 – Процес навчання NBDT

Підсумок. Використовуйте згорткові мережі!

Знаходити ядра, що підходять для певної задачі, можна за допомогою згорткових мереж. Замість того, щоб задавати конкретні числа нашим ядрам, ми призначимо для них параметри, які навчатимуться на даних. У міру навчання ядро все краще і краще фільтруватиме зображення (або карту ознак) для отримання потрібної інформації. Цей автоматичний процес називається навчання ознак. Він узагальнюється для будь-якої нової задачі: нам треба просто заново навчити нашу мережу та знайти відповідні фільтри. Саме тому згорткові мережі такі потужні – жодних проблем із проектуванням ознак.

Зазвичай ми навчаємо не одне ядро, а ієрархію кількох ядер одночасно. Наприклад, застосування ядра 32 на 16 на 16 до зображення 256 на 256 дасть 32 карти ознак розміром 241 на 241 (це стандартне співвідношення: розмір зображення – розмір ядра + 1). Таким чином, ми автоматично вивчимо 32 нових ознаки з інформацією для нашого завдання. Вони використовуються як вхідні дані для наступного ядра, яке знову застосовує до них фільтр. Як тільки ми вивчимо всю ієрархію, ми просто передаємо її у просту нейромережу, яка поєднає ознаки для класифікації вхідного зображення. Це майже все, що потрібно знати про згорткові мережі на концептуальному рівні.

Зробимо підсумок, зведши все в систематичну оцінку CNN-модулів. Що вигідно робити та використовувати, принаймні з початку розробки моделі:

- Використовуйте нелінійність ELU без пакетної нормалізації (batchnorm) або ReLU з нормалізацією.
- Застосовувати вивчену трансформацію колірного простору RGB.
- Використовувати політику лінійного зменшення швидкості навчання (*linear learning rate decay policy*).
- Використовувати суму середнього та максимального pooling-шару.
- Використовувати міні-пакет розміром 128 або 256. Якщо для відеокарти цього занадто багато, зменшуйте швидкість навчання пропорційно розміру пакета.
- Використовувати повнозв'язкові шари як згорткові та усереднювати прогнози для видачі фінального рішення.
- Якщо збільшуєте розмір навчального датасету, перевірте, чи не досягли плато в навчанні. Чистота даних важливіша за розмір.
- Якщо не вдається збільшити розмір вхідного зображення, зменшуйте страйд у наступних шарах, ефект буде приблизно таким самим.
- Якщо ваша мережа має складну і високо оптимізовану архітектуру, як у GoogLeNet, то модифікуйте її з обережністю.

ТЕМА №9: РЕКУРЕНТНІ НЕЙРОННІ МЕРЕЖІ.

Рекурентні нейронні мережі (Recurrent Neural Networks, RNNs) – це популярні моделі, що використовуються для обробки часових залежностей, зокрема для обробки природної мови (NLP), даних сенсорів, тощо. На прикладі NLP завдань, можна сказати, що, по-перше, вони оцінюють довільні речення на основі того, як часто вони зустрічалися в текстах. Це дає нам міру граматичної та семантичної коректності. Такі моделі використовуються, наприклад, у машинному перекладі. По-друге, мовні моделі генерують новий текст. Навчання моделі на поемах Шекспіра дозволить генерувати новий текст, схожий на Шекспіра. Але це тільки дуже прості приклади, які відносяться до NLP. Я же працюють ці моделі, і чому вони такі ефективні для обробки часових рядів? [61-90]

Що таке рекурентні нейронні мережі?

Ідея RNN полягає у послідовному використанні інформації. У традиційних нейронних мережах мається на увазі, що всі входи і виходи незалежні. Але для багатьох завдань це не підходить. Якщо, наприклад, ви хочете передбачити наступне слово у реченні, краще враховувати попередні слова. RNN називаються рекурентними, тому що вони виконують одну і ту ж задачу для кожного елемента послідовності, причому вихід залежить від попередніх обчислень. Ще одна інтерпретація RNN – це мережі, які мають «пам'ять», яка враховує попередню інформацію. Теоретично RNN можуть використовувати інформацію у доволі довгих послідовностях, але на практиці вони обмежені лише кількома кроками (докладніше про це пізніше) [61-65].

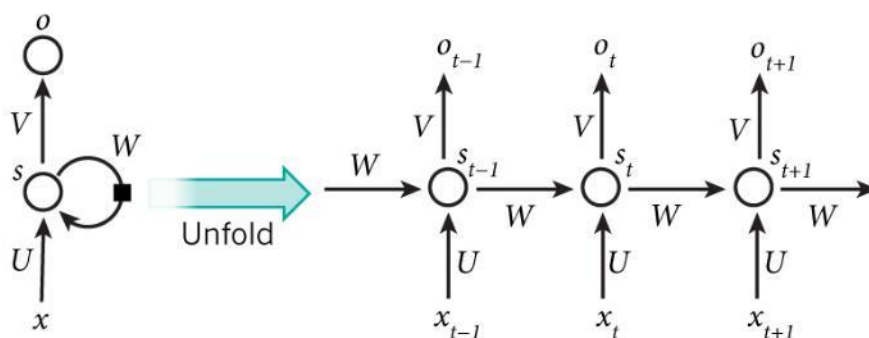


Рисунок 9.1 – Рекурентна нейронна мережа та її розгортка (unfolding)

На рисунку 9.1 показано, що RNN розгортається у повну мережу. Розгорткою ми просто виписуємо мережу для повної послідовності. Наприклад, якщо послідовність є реченням з 5 слів, розгортка буде

складатися з 5 шарів, по шару на кожне слово. Формули, що задають обчислення RNN наступні:

- x_t – це вхід на часовому кроці t . Наприклад, x_1 може бути вектором з одним гарячим станом (*one-hot vector*), що відповідає другому слову речення.
- s_t – це прихований стан на кроці t . Це пам'ять мережі. s_t залежить, як функція, від попередніх станів та поточного входу x_t : $s_t = f(Ux_t + Ws_{t-1})$. Функція f зазвичай нелінійна, наприклад, $\tanh$ або ReLU . s_{-1} , яке потрібне для обчислення першого прихованого стану, зазвичай ініціалізується нулем (нульовим вектором).
- o_t – це вихід на кроці t . Наприклад, якщо ми хочемо передбачити слово у реченні, вихід може бути вектором ймовірностей у нашому словнику: $o_t = \text{softmax}(Vs_t)$

Декілька нотаток:

- Можна інтерпретувати s_t як пам'ять мережі. s_t містить інформацію про те, що сталося на попередніх кроках часу. Вихід o_t обчислюється виключно з урахуванням «пам'яті» s_t . На практиці дещо складніше: s_t не може містити інформацію про занадто велику кількість попередніх кроків;
- На відміну від традиційної глибокої нейронної мережі, яка використовує різні параметри на кожному шарі, RNN має однакові (U, V, W) всіх етапах. Це відбиває той факт, що ми виконуємо те саме завдання на кожному кроці, використовуючи лише різні входи. Це значно зменшує загальну кількість параметрів, які потрібно підібрати;
- Діаграма на рисунку 9.1 має виходи на кожному кроці, але, залежно від завдання, вони можуть не знадобитися. Наприклад при визначенні емоційного забарвлення речення, доцільно дбати лише про кінцевий результат, а не забарвлення після кожного слова. Аналогічно, нам може не знадобитися введення даних на кожному кроці. Основною особливістю RNN є прихований стан, який містить деяку інформацію про послідовність.

Де використовують рекурентні нейромережі?

Рекурентні нейронні мережі продемонстрували великий успіх у багатьох завданнях NLP. На цьому етапі слід згадати, що типом RNN, що найчастіше використовується, є LSTM, які набагато краще захоплюють (зберігають) довгострокові залежності, ніж RNN. LSTM – це, по суті, те ж саме, що і RNN, які будуть розібрані в цьому розділі, вони просто мають

інший спосіб обчислення прихованого стану. Ось деякі приклади додатків RNN до NLP [61-75]

- Мовне моделювання та генерація текстів

З огляду на послідовність слів ми хочемо передбачити ймовірність кожного слова (у словнику). Мовні моделі дозволяють нам виміряти ймовірність вибору, що є важливим вкладом у машинний переклад (оскільки пропозиції з великою вірогідністю є правильними). Побічним ефектом такої здатності є можливість генерувати нові тексти шляхом вибору вихідних ймовірностей. Ми можемо генерувати й інші речі, залежно від того, що собою представляють наші дані. У мовному моделюванні наш вхід зазвичай є послідовністю слів (наприклад, закодованих як вектор з одним гарячим станом (*one-hot*)), а вихід – це послідовність передбачуваних слів. При навчанні нейронної мережі ми подаємо на вхід наступному шару попередній вихід $o_t = x_{t+1}$, оскільки хочемо, щоб результат на кроці t був наступним словом.

- Машинний переклад

Машинний переклад схожий на мовне моделювання, оскільки вектор вхідних параметрів є послідовністю слів вихідною мовою (наприклад, німецькою). Ми хочемо отримати послідовність слів цільовою мовою (наприклад, англійською). Ключова відмінність полягає в тому, що ми отримуємо цю послідовність тільки після того, як побачимо всі вхідні параметри, оскільки перше слово пропозиції може вимагати інформації щодо всієї послідовності слів, що вводяться (рисунок 9.2).

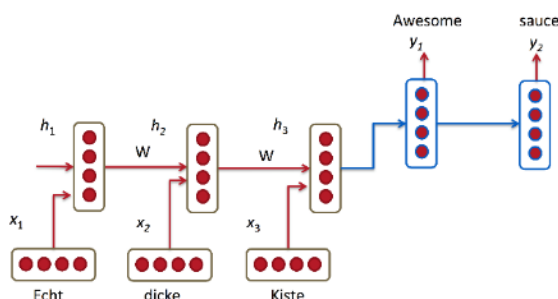


Рисунок 9.2 – Приклад RNN моделі для машинного перекладу

- Розпізнавання мови

За вхідною послідовністю акустичних сигналів від звукової хвилі ми можемо передбачити послідовність фонетичних сегментів разом зі своїми ймовірностями.

- Генерація опису зображень

Разом із згортковими нейронними мережами RNN можуть використовуватися як частина моделі генерації описів нерозмічених зображень. Комбінована модель поєднує згенеровані слова із ознаками, знайденими на зображеннях (рисунок 9.3).

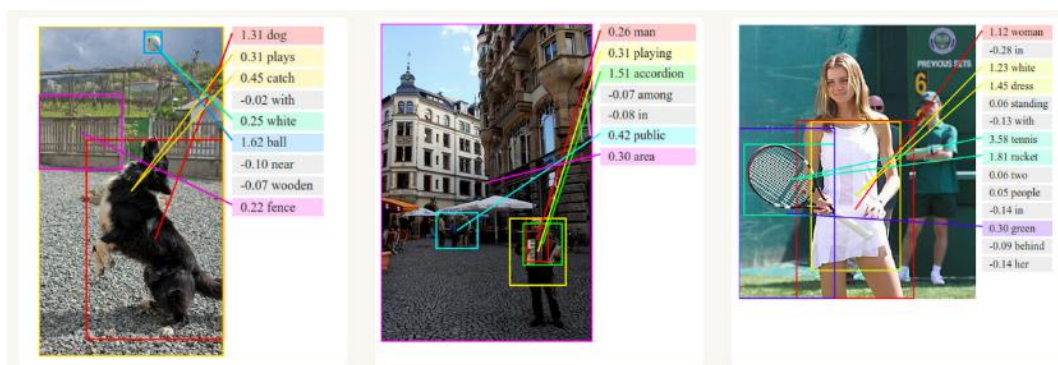


Рисунок 9.3 – Глибокі візуально-семантичні поєднання для створення опису зображень. Джерело:

<http://cs.stanford.edu/people/karpathy/deepimagesent/>

Навчання RNN

Навчання RNN аналогічне до навчання звичайної нейронної мережі. Ми також використовуємо алгоритм зворотного розповсюдження помилки (backpropagation), але з невеликою зміною. Оскільки одні й самі параметри використовуються на всіх часових етапах у мережі, градієнт на кожному виході залежить не тільки від розрахунків поточного кроку, а й від попередніх часових кроків. Наприклад, щоб обчислити градієнт при $t = 4$ нам потрібно було б «поширити помилку» на 3 кроки і підсумовувати градієнти. Цей алгоритм називається "алгоритмом зворотного поширення помилки крізь час" (Backpropagation Through Time, BPTT). Рекурентні нейронні мережі, що пройшли навчання з BPTT, мають труднощі з вивченням довгострокових залежностей (наприклад, залежність між кроками, які знаходяться далеко один від одного) через загасання/підривання градієнта. Щоб уникнути цих проблем, існує певний механізм, а саме були розроблені спеціальні архітектури RNN (наприклад LSTM).

Модифікації RNN

Протягом багатьох років дослідники розробляли складніші типи RNN, щоб упоратися з деякими недоліками класичної моделі.

Двонаправлені рекурентні нейронні мережі (Bidirectional RNNs) засновані на тій ідеї, що вихід у момент часу t може залежати не тільки від попередніх елементів у послідовності, а й від майбутніх. Наприклад, якщо

ви хочете передбачити недостатнє слово в послідовності, враховуючи як лівий, так і правий контекст. Двонаправлені рекурентні нейронні мережі досить прості. Це лише дві RNN, покладені одна на одну. Потім вихід обчислюється з урахуванням прихованого стану обох RNN (рисунок 9.4).

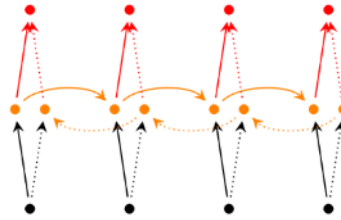


Рисунок 9.4 – Схематичне представлення Bidirectional RNNs

Глибинні рекурентні нейронні мережі схожі на двоспрямовані RNN, тільки тепер у нас є кілька рівнів на кожен крок часу. На практиці це дасть більш високий потенціал, але нам також знадобиться багато даних для навчання.

Мережі LSTM досить популярні в наші дні, ми коротко говорили про них. LSTM не має принципових відмінностей з точки зору архітектури від RNN, але вони використовують іншу функцію для обчислення прихованого стану. Пам'ять в LSTM називається комірками, і ви можете розглядати їх як чорні ящики, які приймають у якості вхідних даних попередній стан h_{t-1} і поточний вхідний параметр x_t . Усередині ці комірки вирішують, яку пам'ять зберегти і яку стерти. Потім вони поєднують попередній стан, поточну пам'ять та вхідний параметр. Виявляється, ці типи одиниць дуже ефективні у захопленні (зберіганні) довгострокових залежностей.

LSTM –мережа з довгою короткостроковою пам'яттю

Розберемося детальніше із LSTM, оскільки ця архітектура є найбільш вживаним різновидом RNN на даний час.

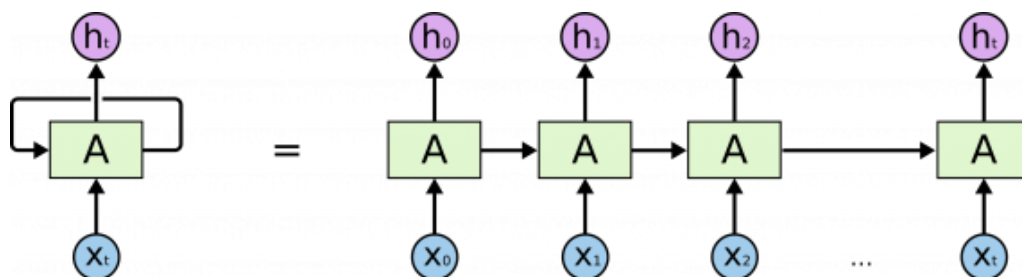
Мотивація створення LSTM

Люди не запускають розумовий процес із нуля у кожний момент часу. Читаючи статтю, ви розумієте значення кожного слова на основі значень попередніх слів. Думки мають властивість накопичуватися та впливати один на одного. Цей принцип використовується в мережах LSTM [70-90].

Прості нейронні мережі не можуть цього зробити, і це є серйозним недоліком. Уявіть, що ви хочете в реальному часі класифікувати події у

фільмі. Неясно, як звичайна нейронна мережа може використовувати знання попередніх подій, щоб вивчити наступні.

Рекурентні нейронні мережі вирішують цю проблему, завдяки присутності циклів, що зберігають інформацію (рисунок 9.5).



Цикли в RNN

На рисунку 9.5 частина нейронної мережі A приймає вхідний сигнал x та виводить значення h . Цикл дозволяє передавати інформацію з одного кроку мережі на інший.

Через наявність циклів RNN виглядають загадково. Однак, якщо подумати, стає зрозуміло, що вони не відрізняються від простої нейронної мережі. RNN можна представити як кілька копій однієї й тієї ж мережі, кожна із яких передає повідомлення наступнику.

Розгорнута структура RNN демонструє, що нейронні мережі, що повторюються, тісно пов'язані з послідовностями і списками. Це природна архітектура нейронної мережі, яка використовується для даних такого типу.

Проблема довгострокових зв'язків

Рекурентна нейронна мережа використовує отриману раніше інформацію для вирішення наступних завдань, наприклад, отримані відеофрагменти для аналізу наступних.

Іноді для вирішення завдання потрібно переглянути лише останню інформацію. Наприклад, побудувати лінгвістичну модель, яка намагається передбачити наступне слово, ґрунтуючись на попередніх. Якщо ми намагаємося передбачити останнє слово у словосполученні «хмари у небі», нам не потрібний додатковий контекст – очевидно, що наступне слово буде «небі». У тих випадках, коли розрив між попередньою інформацією та місцем, у якому вона потрібна, невеликий, RNN упорається із завданням (рисунок 9.6).

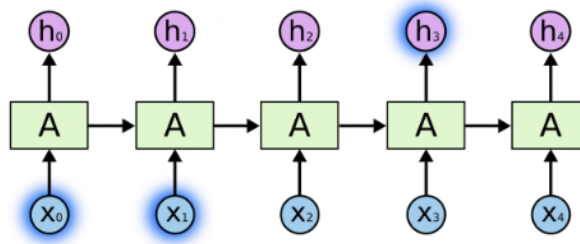


Рисунок 9.6 – Приклад невеликої відстані між поточним станом та передбаченням рекурентної нейронної мережі. Ми знаходимося у місці h_3 .

Але іноді потрібно більше контексту. Розглянемо спробу передбачити останнє слово у тексті «Я виріс у Франції... Я вільно говорю ...? (французькою)». З попередніх слів зрозуміло, що наступним словом, ймовірно, буде назва мови, але якщо ми хочемо назвати правильну мову, потрібно врахувати згадку про Францію. Розрив між необхідною для обліку інформацією та точкою, в якій вона потрібна, стає більшим (рисунок 9.7). На жаль, зі збільшенням цього розриву звичайна рекурентна мережа втрачає зв'язок між інформацією.

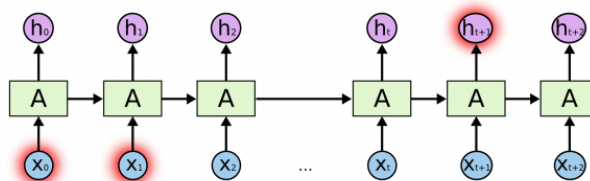


Рисунок 9.7 – Приклад великої відстані між поточним станом та передбаченням.

Теоретично RNN здатні впоратися з такими «довгостроковими залежностями». Дослідник може ретельно підібрати параметри мережі для усунення цієї проблеми. На жаль, на практиці RNN не здатні вирішити це завдання, у ряді роботи було виявлено фундаментальні обмеження RNN.

На щастя, LSTM не має цієї проблеми!

Мережі LSTM

LSTM (long short-term memory, дослівно (довга короткострокова пам'ять) – це тип рекурентної нейронної мережі, здатний навчатися довгостроковим залежностям. LSTM були представлені в роботі [89], згодом удосконалені та поширені іншими дослідниками, добре справляються зі многими завданнями і досі широко застосовуються.

LSTM спеціально розроблені для усунення проблеми довгострокової залежності. Їхня спеціалізація – запам'ятовування інформації протягом тривалих періодів часу, тому їх практично не потрібно навчати!

Всі рекурентні нейронні мережі мають форму ланцюжка модулів нейронної мережі, що повторюються. У стандартних RNN цей модуль, що повторюється, має просту структуру, наприклад, один шар $\tanh$ (рисунок 9.8).

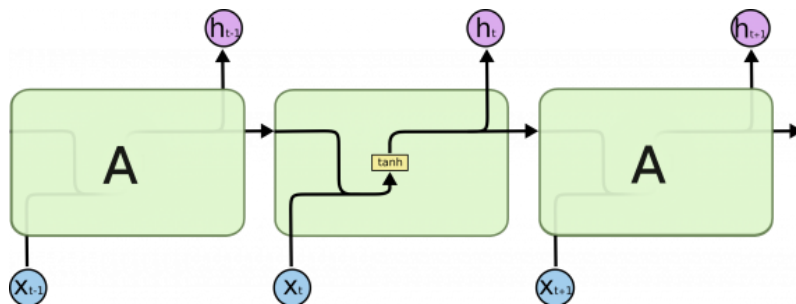


Рисунок 9.8 – Модуль стандартної RNN, що повторюється, складається з одного шару

Модуль LSTM, що повторюється, складається з чотирьох взаємодіючих шарів (рисунок 9.9).

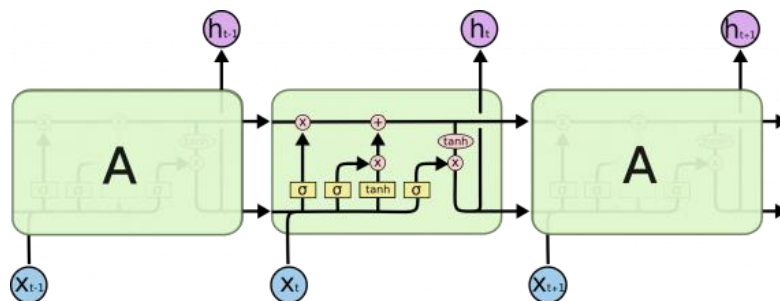


Рисунок 9.9 – Модуль стандартної LSTM, що повторюється, складається з одного шару

Позначення, які ми будемо далі використовувати наведені на рисунку 9.10.

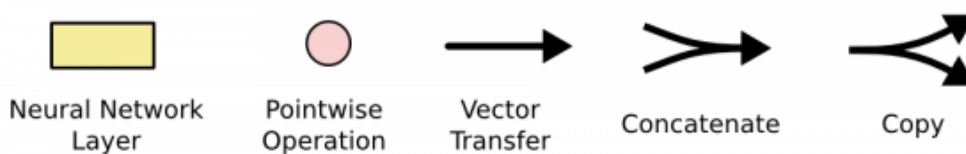


Рисунок 9.10 – Загальні позначення для опису RNN

На рисунках 9.8-9.9 кожна лінія є вектором. Рожеве коло означає крапкові операції, наприклад, підсумовування векторів. Під жовтими комірками розуміються шари нейронної мережі. Пос'єднання ліній є об'єднання векторів, а знак розгалуження – копіювання вектора з подальшим зберіганням у різних місцях.

Принцип роботи мережі LSTM

Ключовим поняттям LSTM є стан комірки: горизонтальна лінія, що проходить через верхню частину діаграми (рисунок 9.11).

Стан комірки нагадує конвеєрну стрічку. Воно проходить через весь ланцюжок, піддаючись незначним лінійним перетворенням.

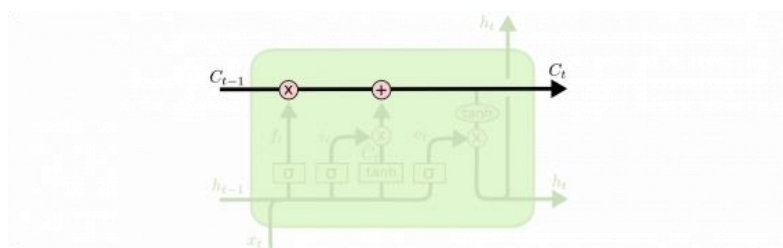


Рисунок 9.11 – Стан LSTM комірки

У LSTM можна зменшувати чи збільшувати кількість інформації у стані комірки, залежно від потреб. Для цього використовуються структури, що ретельно налаштовуються, звані «гейтами».

Гейт – це «ворота», що пропускають або не пропускають інформацію. Гейти складаються з сигмовидного шару нейронної мережі та операції поточкового множення (рисунок 9.12).

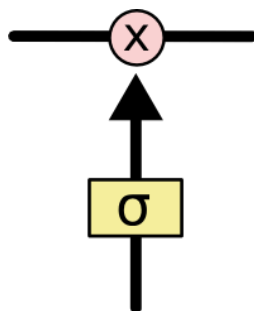


Рисунок 9.12 – Схематичне представлення гейту

На виході сигмовидного шару видаються числа від нуля до одиниці, визначаючи скільки відсотків кожної одиниці інформації пропустити далі.

Значення "0" означає "не пропустити нічого", значення "1" – "пропустити все".

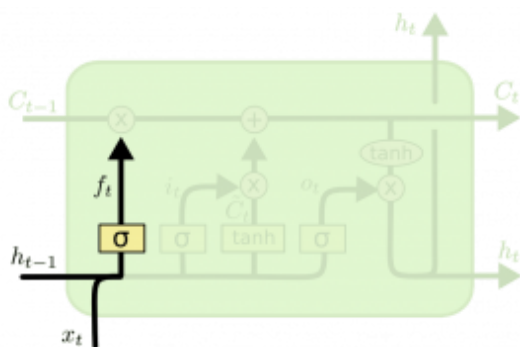
Покрокова схема роботи мережі LSTM

LSTM має три такі гейти для контролю стану комірки.

Шар втрати

На першому етапі LSTM потрібно вирішити, яку інформацію ми збираємось викинути зі стану комірки. Це рішення приймається сигмовидним шаром, званим шаром *гейту втрати*. Він отримує на вхід h і x і видає число від 0 до 1 для кожного номера в стані комірки C . 1 означає "повністю зберегти", а 0 – "повністю видалити" (рисунок 9.13).

Повернемося до прикладу лінгвістичної моделі. Спробуємо передбачити таке слово, що ґрунтується на всіх попередніх. У такому завданні стан комірки включає мовний рід предмета, щоб використовувати правильні займенники. Коли з'являється новий предмет, потрібно забути рід попереднього предмету.



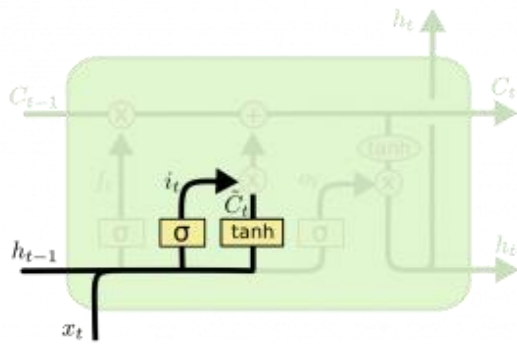
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Рисунок 9.13 – Визначення гейту втрат

Шар збереження

На наступному кроці потрібно вирішити, яку нову інформацію зберегти в стан комірки. Розіб'ємо процес на дві частини. Спочатку сигмоїдний шар, званий *шаром гейту входу або оновлення*, вирішує, які значення *потрібно оновити*. Потім шар $\tanh$ створює вектор нових значень-кандидатів C , які додаються до стану. На наступному етапі ми об'єднаємо ці два значення для оновлення стану (рисунок 9.14).

У прикладі лінгвістичної моделі ми хотіли б додати рід нового предмета у стан комірки, щоб замінити їм рід старого.



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

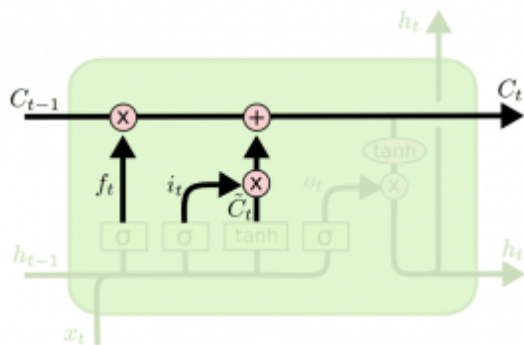
Рисунок 9.14 – Визначення гейту збереження

Новий стан

Тепер оновимо попередній стан комірки для отримання нового стану C . Спосіб оновлення вибраний, тепер реалізуємо саме оновлення.

Помножимо старий стан на f , втрачаючи інформацію, яку вирішили забути. Потім додаємо $i * C$. Це нові значення кандидатів, які масштабуються залежно від того, як ми вирішили оновити кожне значення стану (рисунок 9.15).

У випадку з лінгвістичною моделлю ми відкинемо інформацію про рід старого суб'єкта і додаємо нову інформацію.

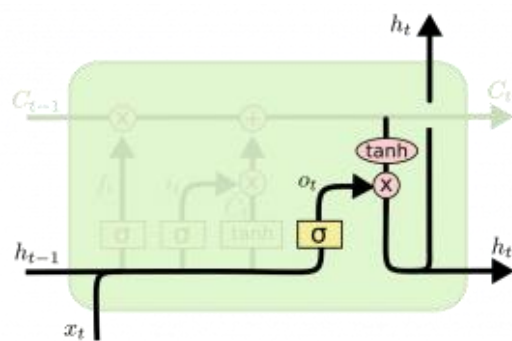


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Рисунок 9.15 – Визначення нового стану

Зрештою, треба вирішити, що хочемо отримати на виході. Результат буде відфільтрованим станом комірки. Спочатку запускаємо сигмоїдний шар, який вирішує, які частини стану комірки виводити. Потім пропускаємо стан комірки через $\tanh$ (щоб розмістити всі значення в інтервалі $[-1, 1]$) та множимо його на вихідний сигнал сигмоподібного гейту (рисунок 9.16).

Для лінгвістичної моделі, оскільки мережа працювала лише з предметом, вона може вивести інформацію, що відноситься до дієслова. Наприклад, мережа виведе інформацію про те, в якому числі представлено предмет (єдине або множинне) для правильного відмінювання дієслова.



$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

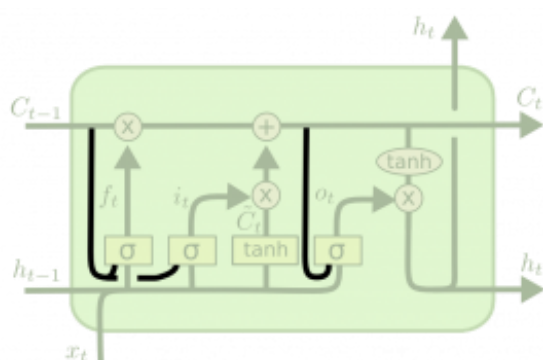
$$h_t = o_t * \tanh(C_t)$$

Рисунок 9.16 – Визначення виходу з комірки

Приклади LSTM

Описана вище схема традиційна для LSTM. Але не всі LSTM ідентичні. Насправді майже в кожній статті використовуються версії, що відрізняються. Відмінності незначні, але варто згадати деякі з них.

У популярному варіанті LSTM, представленому в [88], дозволяється шарам гейтів переглядати стан комірки (рисунок 9.17)



$$f_t = \sigma(W_f[C_{t-1}, h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i[C_{t-1}, h_{t-1}, x_t] + b_i)$$

$$o_t = \sigma(W_o[C_t, h_{t-1}, x_t] + b_o)$$

Рисунок 9.17 – Визначення LSTM з роботи Gers & Schmidhuber

На рисунку 9.17 «глазок» є у всіх гейтів, але в багатьох статтях він є лише в деяких гейтів.

Інший варіант – використання пов'язаних гейтів втрати та входу. Замість того, щоб окремо вирішувати, що забути, а до чого додати нову інформацію, ми ухвалюємо ці рішення одночасно. Ми забуваємо інформацію лише тоді, коли потрібно помістити щось нове на тому самому місці. Нові значення вносяться в стан лише тоді, коли ми забуваємо щось старіше (рисунок 9.18).

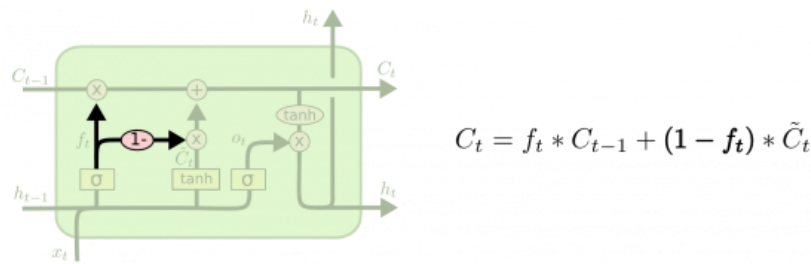


Рисунок 9.18 – різновид LSTM з використанням пов'язаних гейтів втрати та входу

Дещо відрізняється від LSTM – **Gated Recurrent Unit або GRU**, введений в [91]. GRU поєднує гейти втрати та входу в єдиний «шлюз оновлення». У ньому також об'єднуються стан комірки та прихований стан та вносяться деякі інші зміни (рисунок 9.19). Модель, що виходить, простіша в порівнянні зі стандартними типами LSTM і стає все більш популярною.

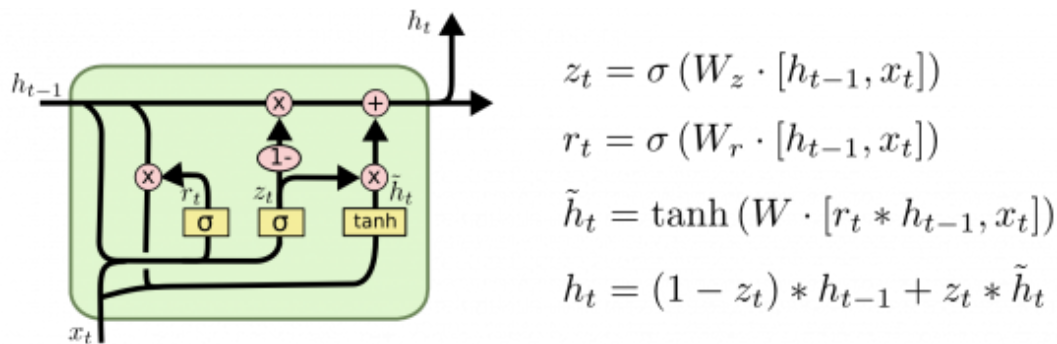


Рисунок 9.19 – приклад GRU архітектури

Це лише деякі з найвідоміших варіантів LSTM. Є багато інших, таких як RNN з гейтом глибини [92]. Відомий також зовсім інший підхід до вирішення довгострокових залежностей, наведений у [93].

Який із перерахованих варіантів найкращий? Чи велике значення відмінності? У [81] наведено широке порівняння популярних версій LSTM, а також продемонстровано, що вони приблизно однакові. У [94] випробувано понад десять тисяч архітектур RNN. Деякі з них працюють краще за LSTM при вирішенні певних завдань.

LSTM були великим кроком у розвитку RNN. Природно запитати себе: а що, можна піти далі? Загальна думка для дослідників: «Так! Наступний крок полягає у використанні механізму уваги! Ідея полягає в тому, щоб RNN на кожному кроці вибирала інформацію для перегляду з більшої кількості даних. Наприклад, якщо ви використовуєте RNN для

створення опису зображення, мережа може вибрати частину зображення для перегляду кожного слова. Саме це і було зроблено в [95] і стало відправною точкою для вивчення механізму уваги (опис цього механізму виходить за межі цього посібника).

GRU – керований рекурентний нейрон

Розглянемо також трохи детальніше GRU архітектуру.

Керований рекурентний нейрон з'явився у 2014 році [80], тоді як мережі довгої короткострокової пам'яті – 1997-го.

GRU можна розглядати як простішу версію мереж довгої короткострокової пам'яті. Він включає в себе багато подібних понять, але має куди менше властивостей, а тому при тому самому розмірі прихованого шару навчається швидше. Нещодавні дослідження також показали, що керований рекурентний нейрон та мережі довгої короткострокової пам'яті мають порівнянну точність, а в деяких випадках GRU навіть точніше. Таким чином, немає суворого правила, яке вказує, що саме вибирати. Це як вибір найкращої функції активації для звичайних нейронних мереж – потрібно спробувати різні варіанти та подивитися, що краще працює для ваших конкретних даних.

Опишемо детальніше будову керованого рекурентного нейрона (рисунок 9.19). Насамперед будемо все розглядати частинами. Уявімо собі вміст між попереднім і наступним шарами у вигляді «чорної скриньки». У найпростішій нейронній мережі прямого поширення ця «чорна скринька» містить лише деяку нелінійну функцію, наприклад гіперболічний тангенс або Relu. У простій рекурентній мережі ми просто під'єднуємо вихід із «чорної скриньки» до нього із затримкою в часі в мінус одиницю. У *оновлюваній рекурентній* мережі є ще операція оновлення між виходом простої рекурентної мережі та попереднім вихідним значенням. Цю операцію можна визначити як *логічний вентиль (gate)*. Оскільки він приймає значення від 0 до 1, а другий вентиль повинен дорівнювати одиниця мінус це значення, то це вентиль, що вибирає між двома варіантами: або прийняти старе значення, або прийняти нове. В результаті ми отримуємо суміш їх обох.

Переходимо тепер до власне керованого рекурентного нейрона, GRU (рисунок 9.19). Зауважте, тут немає жодної нової математики – лише те, що ми вже вивчали про матриці вагових коефіцієнтів, помножених на вхідні дані та пропущені через нелінійні функції та вентиля. Ми знову бачимо оновлення z_t . У нас також є Γ_t – вентиль скидання. Він має точно таку ж

функціональну форму, що і вентиль оновлення, і всі його вагові коефіцієнти мають той же розмір, але його положення в нашому «чорному ящику» дещо відрізняється. Вентиль скидання множиться на попереднє значення прихованого стану. Він контролює, що з попереднього прихованого стану враховуватиметься при створенні нового кандидата значення прихованого стану. Інакше кажучи, він може скинути приховане значення. Якщо $r_t = 0$, то

$$\tilde{h}_t = g(x_t W_{xh} + b_h)$$

Це ніби x_t було початком нової послідовності.

Зверніть увагу, що це неповна картина, оскільки є лише кандидатом у нові h_t , оскільки саме нове h_t буде комбінацією і h_{t-1} , що керується вентилем оновлення z_t .

Нарешті, рисунок 9.20 наводить порівняння трьох розглянутих архітектур RNN, GRU, LSTM на одному зображенні.

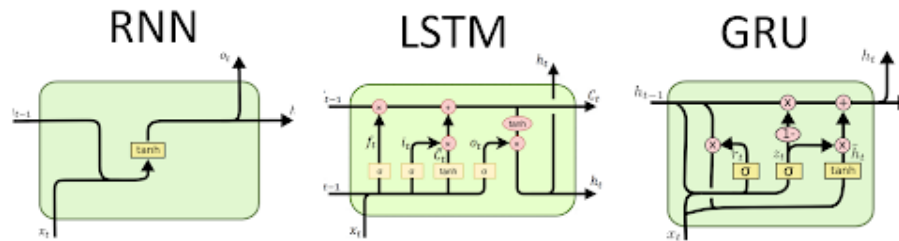


Рисунок 9.20 – Порівняння трьох розглянутих архітектур RNN, GRU, LSTM

ТЕМА №10: ГЕНЕРАТИВНО-ЗМАГАЛЬНА НЕЙРОМЕРЕЖА (GAN)

Загальні положення

Генеративно-змагальна нейромережа (Generative adversarial network, GAN) – це архітектура, що складається з генератора та дискримінатора, налаштованих на роботу один проти одного. Звідси GAN і отримала назву генеративно-змагальна у разі роботи із зображеннями, у всьому іншому – це згорткова нейронна мережа.

GAN були вперше представлені в роботі 2014 року Ian'a Goodfellow'a та інших дослідників університету Монреаля, включаючи Yoshua Bengio. Директор Facebook з досліджень штучного інтелекту Yann LeCun назвав змагальне тренування мереж "найцікавішою ідеєю в машинному навчанні за останні 10 років" під час їх появи.

Навряд чи ви подумаєте, що програміста можна назвати артистом, але насправді програмування це дуже творча професія. Творчість, заснована на логіці – як казав John Romero.

Потенціал GAN величезний, оскільки вони імітують будь-який розподіл даних. GAN навчають створювати структури, що страхотливо схожі на сутності з нашого світу в області зображень, музики, мови, прози. Генеративно-змагальні мережі, у певному сенсі, роботи-художники, і результат їхньої роботи вражає (рисунок 10.1) [96-101].



Рисунок 10.1 – Реалістичні зображення неіснуючих знаменитостей, створені дослідниками за допомогою GAN

Дискримінатор та генератор.

Дискримінатор.

Дискримінаційні алгоритми намагаються класифікувати вхідні дані. З огляду на особливості отриманих даних вони намагаються визначити категорію, до якої вони належать.

Наприклад, пробігаючи всі слова у листі дискримінаційний алгоритм може передбачити, чи є повідомлення спамом чи неспамом. Спам – це категорія, а пакет слів, зібраний з електронної пошти – образи, які становлять вхідні дані. Математично категорії позначають y , а образи позначають x . Запис $p(y|x)$ використовується для позначення "ймовірності y при заданому x ", яка позначає "ймовірність того, що електронний лист є спамом при наявному наборі слів".

Отже, дискримінаційні функції зіставляють образи із категорією. Вони зайняті лише цією кореляцією.

Генератор

Генеративні алгоритми зайняті зворотним. Замість того, щоб передбачати категорію за наявними образами, вони намагаються підібрати образи до цієї категорії.

У той час як дискримінаційні алгоритми хвилює взаємозв'язок між y та x , генеративні алгоритми хвилює "звідки беруться x ". Вони дозволяють знаходити $p(x|y)$, ймовірність x при даному y або ймовірність образів при даному класі (генеративні алгоритми також можуть використовуватися як класифікатори. Вони можуть робити більше, ніж класифікувати вхідні дані.)

Ще одне уявлення про роботу генеративних алгоритмів можна отримати, розділяючи дискримінаційні моделі від генеративних таким чином:

- Дискримінаційні моделі вивчають межу між класами;
- Генеративні моделі моделюють розподіл окремих класів.

Як працюють GAN

Одна нейронна мережа, звана генератором, генерує нові екземпляри даних, інша – дискримінатор, оцінює їх справжність; тобто дискримінатор вирішує, чи кожен екземпляр даних, який він розглядає, відноситься до набору тренувальних даних чи ні.

Припустимо, ми намагаємося зробити щось банальніше, ніж повторити портрет Мони Лізи. Ми згенеруємо рукописні цифри, подібні до

тих, що є в наборі даних MNIST. Мета дискримінатора – розпізнати справжні екземпляри із набору.

Тим часом генератор створює нові зображення, які він передає дискримінатору. Він робить це, сподіваючись, що вони будуть прийняті справжніми, хоча є підробленими. Ціль генератора полягає в тому, щоб генерувати рукописні цифри, які будуть пропущені дискримінатором. Мета дискримінатора – визначити, чи є зображення справжнім.

Кроки, які проходить GAN (рисунок 10.2):

- Генератор отримує випадкове число та повертає зображення.
- Це згенероване зображення подається дискримінатор поряд з потоком зображень, взятих з фактичного набору даних.
- Дискримінатор приймає як реальні, так і підроблені зображення і повертає ймовірності числа від 0 до 1, причому 1 являє собою справжнє зображення і 0 представляє фальшиве.

Таким чином, є подвійний цикл зворотного зв'язку:

- Дискримінатор перебуває у циклі із достовірними зображеннями.
- Генератор перебуває у циклі разом із дискримінатором

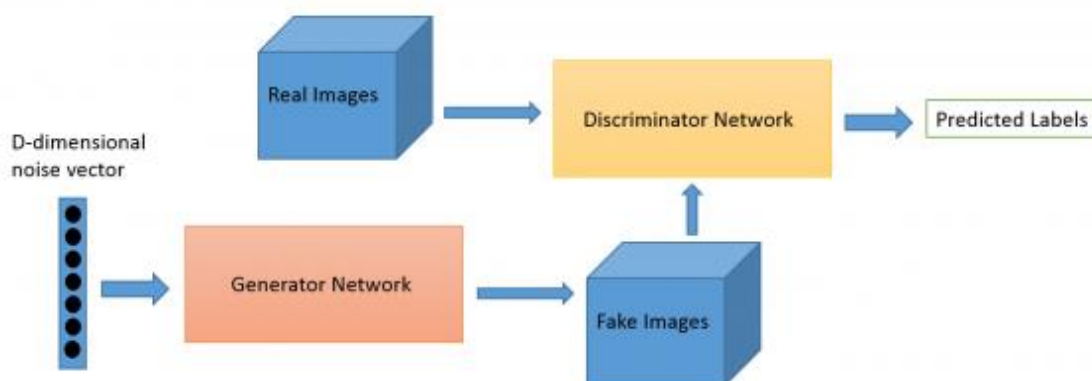


Рисунок 10.2 – Схема роботи GAN

Ви можете уявити GAN як фальшивомонетника та поліцейського, які грають у кішки мишки, де фальсифікатор вчиться виготовляти хибні купюри, а поліцейський вчиться їх виявляти. Обидва динамічні; тобто поліцейський теж тренується (можливо, центральний банк відзначає пропущені купюри), і кожна сторона приходить до вивчення методів іншого у постійній ескалації.

Мережа дискримінаторів являє собою стандартну згорткову мережу, яка може класифікувати зображення, що подаються на неї за допомогою біномного класифікатора, що розпізнає зображення як реальні або як підроблені. **Генератор** в певному сенсі являє собою зворотну згорткову мережу: хоча стандартний згортковий класифікатор приймає зображення і зменшує його роздільну здатність, щоб отримати ймовірність, генератор приймає вектор випадкового шуму і перетворює його в зображення. Перший відсіює дані за допомогою методів зниження дискретизації, таких як maxpooling, а другий генерує нові дані (рисунок 10.3).

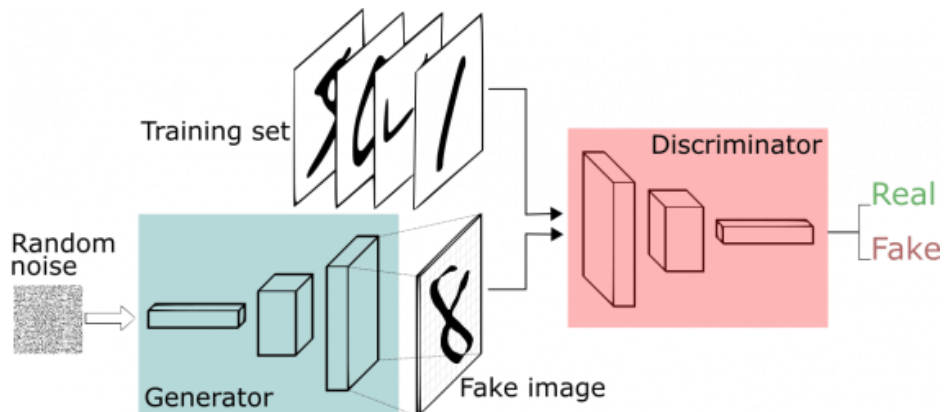


Рисунок 10.3 – Приклад GAN із зображеннями MNIST

Обидві мережі намагаються оптимізувати цільову функцію чи функцію втрат у грі zero-sum. Це, власне, модель актора-критика (actor-critic). Коли дискримінатор змінює свою поведінку, то генератор змінює, і навпаки.

Ще приклад. Коли починається навчання, генератор видає свідомо фальшиві дані, а дискримінатор швидко вчиться визначати, що це фальшивка (рисунок 10.4)



Рисунок 10.4 – Приклад реальних та згенерованих даних

У міру навчання генератор наближається до виходу, який може обдурити дискримінатор (рисунок 10.5)



Рисунок 10.5 – Приклад наближення до реального зображення під час навчання

Зрештою, якщо навчання генератора проходить добре, дискримінатор гірше розпізнає різницю між справжнім та фальшивим. Він починає класифікувати підроблені дані як реальні, і їхня точність знижується (рисунок 10.6).



Рисунок 10.6 – Приклад доброго навчання генератора

Інтуїтивний процес тренування

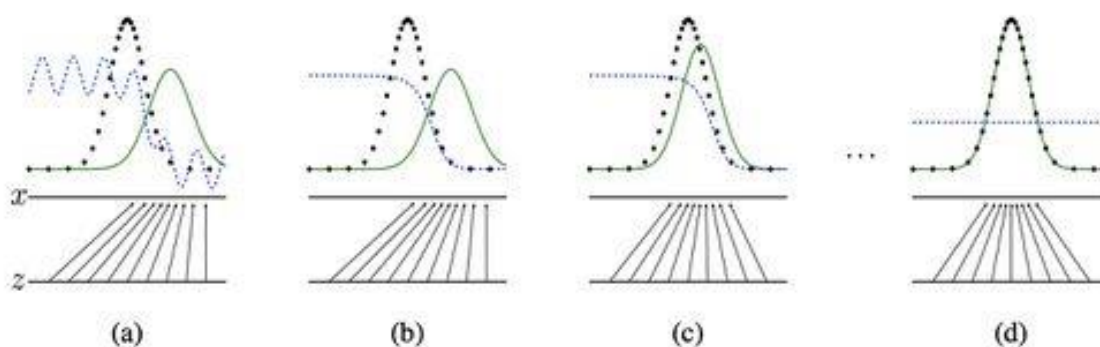


Рисунок 10.7 – Ілюстрація процесу тренування змагальних мереж GAN, що породжують.

Генеративні-змагальні мережі навчаються шляхом одночасного оновлення дискримінаційного розподілу (D синя пунктирна лінія), так щоб дискримінатор міг розрізняти об'єкти з розподілу тренувального сету (чорна пунктирна в точку лінія) та з розподілу генератора (G зелена суцільна лінія). Нижня горизонтальна лінія є область, з якої складена вибірка z , в нашому випадку рівномірно. Горизонтальна лінія над нею є частиною області x . Стрілками на рисунку 10.7 показано, як відображення $x = G(z)$, накладає

нерівномірний розподіл p_g (p_g – імовірнісний розподіл генератора) на тренувальний. G стискається в областях із високою щільністю та розширюється в областях із низькою. Розглянемо описаний на рисунку 10.7 процес. (a) Близька збіжність пари, що змагається: p_g схоже на розподіл p_{data} та D частково-точний класифікатор. (b) У внутрішньому циклі алгоритму D навчається відрізняти об'єкти з тренувальних даних, сходяться до $p_{data}(x)/(p_{data}(x)+p_g(x))$ (c) Після оновлення G градієнт D привів $G(z)$ до пересування в область з більшою ймовірністю бути класифікованим як дані. (d) Після кількох кроків навчання G і D прийдуть у стан, у якому не зможуть поліпшитися, оскільки виконуватиметься умова $p_g = p_{data}$ і дискримінатор не зможе розрізняти два розподіли і його вихід завжди буде $D(x) = 1/2$.

Поліпшення навчання GAN

Більшість GAN'ів схильна до таких проблем:

- Несходимість рішення (non-convergence): параметри моделі дестабілізуються та не сходяться;
- Схлопування мод розподілу (mode collapse): генератор колапсує, тобто видає обмежену кількість різних зразків;
- Зникає градієнт (diminished gradient): дискримінатор стає занадто "сильним", а градієнт генератора зникає і навчання не відбувається;
- Висока чутливість до гіперпараметрів.
- Універсального підходу до їх вирішення немає, але є практичні поради, які можуть допомогти. Основними з них є:
 1. Нормалізація даних. Усі ознаки у діапазоні $[-1;1]$;
 2. Заміна функції помилки для G з $\min \log(1-D)$ на $\max \log D$, тому що вихідний варіант має маленький градієнт на ранньому етапі навчання та великий градієнт при збіжності, а запропонований навпаки;
 3. Семплювання із багатовимірного нормального розподілу замість рівномірного;
 4. Використовувати нормалізаційні шари (наприклад, batch normalization або layer normalization) G і D ;
 5. Використовувати мітки даних, якщо вони є, тобто навчати дискримінатор ще й класифікувати зразки.

Автокодери та VAE

Корисно порівняти генеративні-змагальні мережі з іншими нейронними мережами, такими як автокодери (автоенкодери) та варіаційні автокодери.

Автокодери кодують вхідні дані у вектори. Вони створюють приховане або стисле уявлення необроблених даних. Вони корисні при зменшенні розмірності: вектор, який служить прихованим поданням, стискає необроблені дані в меншу кількість. Автокодери можуть бути пов'язані з так званим декодером, який дозволяє відновлювати вхідні дані на основі їхнього прихованого уявлення, як і у випадку з машиною Больцмана (рисунок 10.8).

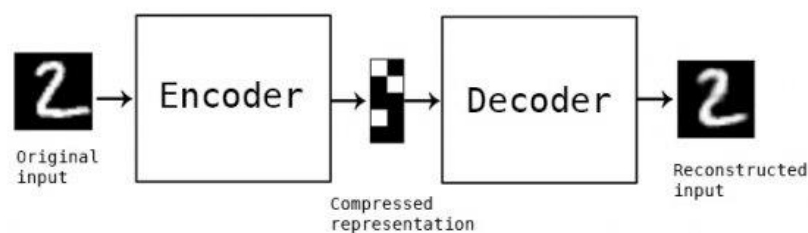


Рисунок 10.8 – Приклад роботи автокодера

Варіаційні автокодери є генеративним алгоритмом, який додає додаткове обмеження для кодування вхідних даних, а саме те, що приховані уявлення нормалізуються. Варіаційні автокодери здатні стискати дані як автокодери та синтезувати дані подібно до GAN. Однак, у той час як GAN генерують дані деталізовано, зображення, створені VAE, є більш розмитими.

Отже, можна розділити генеративні алгоритми на три типи, які мають:

- категорію, що передбачають пов'язані функції (Naive Bayes);
- приховане уявлення, що передбачають пов'язані функції (VAE, GAN);
- деякі образи, що передбачають інше (inpainting, imputation);

Поради щодо навчання GAN

Коли тренуєте дискримінатор, утримуйте значення генератора постійними; і коли ви тренуєте генератор, утримуйте дискримінатор на одному рівні. Кожен має тренуватися проти статичного супротивника.

Наприклад, генератору це дозволить краще зчитувати градієнт, яким він повинен вчитися.

Так само попереднє тренування дискримінатора проти MNIST сприятиме встановленню чіткішого градієнта.

Кожна частина GAN може здолати іншу. Якщо дискримінатор занадто хороший, він повертатиме значення дуже близькі до 0 або до 1, так що генератор зазнає труднощів у читанні градієнта. Якщо генератор занадто хороший, він постійно використовуватиме недоліки дискримінатора, що призводять до неправильних негативів.

GAN вимагає багато часу на тренування. На одному GPU тренування може займати години, а на одному CPU – більше одного дня. Незважаючи на складність налаштування і, отже, використання, GAN стимулювали створення багатьох цікавих досліджень та статей:

- PGGAN – Прогресивна генеративна нейронна мережа NVIDIA
- TwinGAN – внутрішня передача людських портретів
- Xogan – перетворення одного зображення на кілька
- Метод відновлення очей за допомогою зразка GAN з Facebook AI Research

Приклад коду

Ось приклад GAN, запрограмованої в бібліотеці Keras:

```
class GAN():
    def __init__(self):
        self.img_rows = 28
        self.img_cols = 28
        self.channels = 1
        self.img_shape = (self.img_rows, self.img_cols, self.channels)

        optimizer = Adam(0.0002, 0.5)

        # Build and compile the discriminator
        self.discriminator = self.build_discriminator()
        self.discriminator.compile(loss='binary_crossentropy',
                                   optimizer=optimizer,
                                   metrics=['accuracy'])

        # Build and compile the generator
```

```

self.generator = self.build_generator()
self.generator.compile(loss='binary_crossentropy', optimizer=optimizer)

# The generator takes noise as input and generated imgs
z = Input(shape=(100,))
img = self.generator(z)

# For the combined model we will only train the generator
self.discriminator.trainable = False

# The valid takes generated images as input and determines validity
valid = self.discriminator(img)

# The combined model (stacked generator and discriminator) takes
# noise as input => generates images => determines validity
self.combined = Model(z, valid)
self.combined.compile(loss='binary_crossentropy', optimizer=optimizer)

def build_generator(self):

    noise_shape = (100,)

    model = Sequential()

    model.add(Dense(256, input_shape=noise_shape))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(1024))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(np.prod(self.img_shape), activation='tanh'))
    model.add(Reshape(self.img_shape))

    model.summary()

    noise = Input(shape=noise_shape)
    img = model(noise)

    return Model(noise, img)

def build_discriminator(self):

```

```

img_shape = (self.img_rows, self.img_cols, self.channels)

model = Sequential()

model.add(Flatten(input_shape=img_shape))
model.add(Dense(512))
model.add(LeakyReLU(alpha=0.2))
model.add(Dense(256))
model.add(LeakyReLU(alpha=0.2))
model.add(Dense(1, activation='sigmoid'))
model.summary()

img = Input(shape=img_shape)
validity = model(img)

return Model(img, validity)

def train(self, epochs, batch_size=128, save_interval=50):

    # Load the dataset
    (X_train, _), (_, _) = mnist.load_data()

    # Rescale -1 to 1
    X_train = (X_train.astype(np.float32) - 127.5) / 127.5
    X_train = np.expand_dims(X_train, axis=3)

    half_batch = int(batch_size / 2)

    for epoch in range(epochs):

        # -----
        # Train Discriminator
        # -----

        # Select a random half batch of images
        idx = np.random.randint(0, X_train.shape[0], half_batch)
        imgs = X_train[idx]

        noise = np.random.normal(0, 1, (half_batch, 100))

        # Generate a half batch of new images
        gen_imgs = self.generator.predict(noise)

```

```

        # Train the discriminator
        d_loss_real = self.discriminator.train_on_batch(imgs,
np.ones((half_batch, 1)))
        d_loss_fake = self.discriminator.train_on_batch(gen_imgs,
np.zeros((half_batch, 1)))
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # -----
        # Train Generator
        # -----

        noise = np.random.normal(0, 1, (batch_size, 100))

        # The generator wants the discriminator to label the generated samples
        # as valid (ones)
        valid_y = np.array([1] * batch_size)

        # Train the generator
        g_loss = self.combined.train_on_batch(noise, valid_y)

        # Plot the progress
        print ("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" % (epoch, d_loss[0],
100*d_loss[1], g_loss))

        # If at save interval => save generated image samples
        if epoch % save_interval == 0:
            self.save_imgs(epoch)

def save_imgs(self, epoch):
    r, c = 5, 5
    noise = np.random.normal(0, 1, (r * c, 100))
    gen_imgs = self.generator.predict(noise)

    # Rescale images 0 - 1
    gen_imgs = 0.5 * gen_imgs + 0.5

    fig, axs = plt.subplots(r, c)
    cnt = 0
    for i in range(r):
        for j in range(c):
            axs[i,j].imshow(gen_imgs[cnt, :, :, 0], cmap='gray')
            axs[i,j].axis('off')
            cnt += 1

```



```
fig.savefig("gan/images/mnist_%d.png" % epoch)
plt.close()

if __name__ == '__main__':
    gan = GAN()
    gan.train(epochs=30000, batch_size=32, save_interval=200)
```

Деякі різновиди архітектур GAN

Conditional Generative Adversarial Nets, CGAN

Умовні змагальні мережі, що породжують (Conditional Generative Adversarial Nets, CGAN) – це модифікована версія алгоритму GAN, яка може бути сконструйована за допомогою подачі даних y , що є умовою для генератора та дискримінатора [98]. y може бути будь-якою додатковою інформацією, наприклад, міткою класу або даними інших моделей, що може дозволити контролювати процес генерації даних. Наприклад, можна подавати параметр y як умову на клас для генерації чисел, схожих на MNIST.

Як вже було згадано на вхід генератора та дискримінатора з GAN подається додаткова інформація y , наприклад, у випадку з багат шаровими перцептронами умова може бути представлена додатковим вхідним шаром. У генераторі апріорна ймовірність шуму $p(z)$ і умова y комбінуються в об'єднане приховане уявлення, а змагальна модель, що тренує, надає достатньо свободи в тому як це уявлення складається. У дискримінаторі x та y представлені як вхідні параметри (рисунк 10.9).

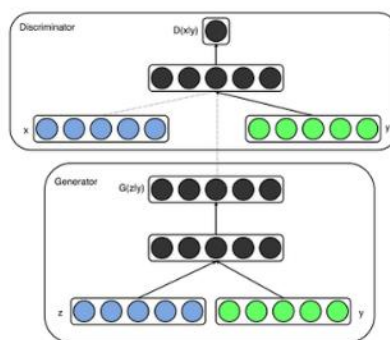


Рисунок 10.9 – Архітектура CGAN

У такому разі завдання оптимізації буде виглядати так:

$$\min_G \min_D E_{x \sim p_{data}} [\log D(x|y)] + E_{z \sim p_z} [\log(1 - D(G(x|y)))]$$

Як приклад використання цього алгоритму можна розглянути завдання генерації рукописних цифр (рисунок 10.10, CGAN був натренований на датасеті MNIST з мітками класів представлених у вигляді one-hot векторів).



Рисунок 10.10 – Цифри, згенеровані з допомогою CGAN

DCGAN (Deep Convolutional Generative Adversarial Nets)

DCGAN – це модифікація алгоритму GAN, в основі якої лежать згорткові нейронні мережі (рисунок 10.11) [99].

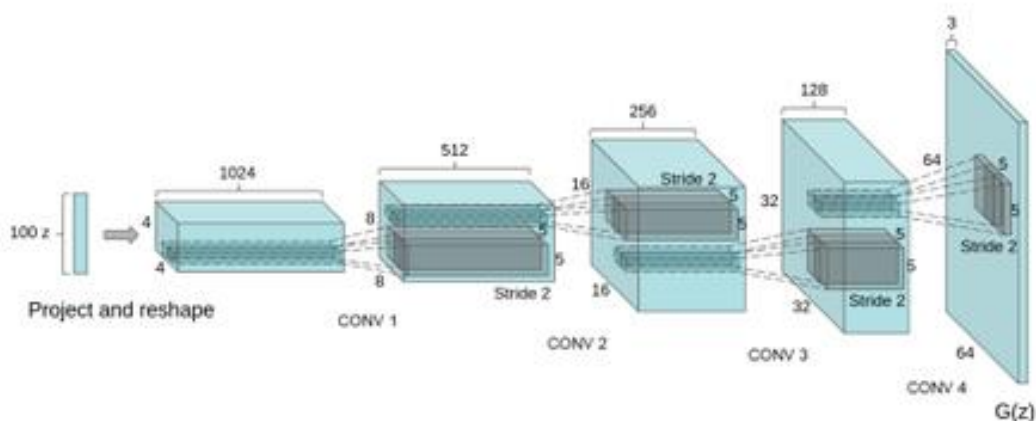


Рисунок 10.10 – Модель генератора у архітектурі DCGAN

Завдання пошуку зручного представлення ознак на великих обсягах не розмічених даних є однією з найбільш активних сфер досліджень, зокрема представлення зображень та відео. Одним із зручних способів пошуку уявлень може бути DCGAN. Використання згорткових нейронних

мереж безпосередньо не давало хороших результатів, тому було внесено обмеження на шари згортки. Ці обмеження лежать в основі DCGAN:

- Заміна всіх пулінгових шарів на страйдингові згортки (strided convolutions) у дискримінаторі та частково-страйдингові згортки (fractional-strided-convolutions) у генераторі, що дозволяє мережам знаходити відповідні зниження та підвищення розмірностей;
- Використання батчингової нормалізації для генератора і дискримінатора, тобто нормалізація входу так, щоб середнє значення дорівнювало нулю і дисперсія дорівнювала одиниці. Не варто використовувати батч-нормалізація для вихідного шару генератора та вхідного дискримінатора.
- Видалення всіх повнозв'язкових прихованих рівнів для глибших архітектур;
- Використання ReLU як функції активації в генераторі для всіх шарів, крім останнього, де використовується tanh;
- Використання LeakyReLU як функція активації в дискримінаторі для всіх шарів.

Крім завдання генерації об'єктів, цей алгоритм добре показує себе у витягуванні ознак. Даний алгоритм був натренований на наборі даних Imagenet-1k, після чого були використані значення зі згорткових шарів дискримінатора, піддані max-pooling', щоб утворити матриці 4 на 4 і отримати загальний вектор ознак на їх основі. L2-SVM, з отриманим уявленням, на наборі даних CIFAR-10 перевищує точність рішення, засновані на алгоритмі K-Means.

StackGAN

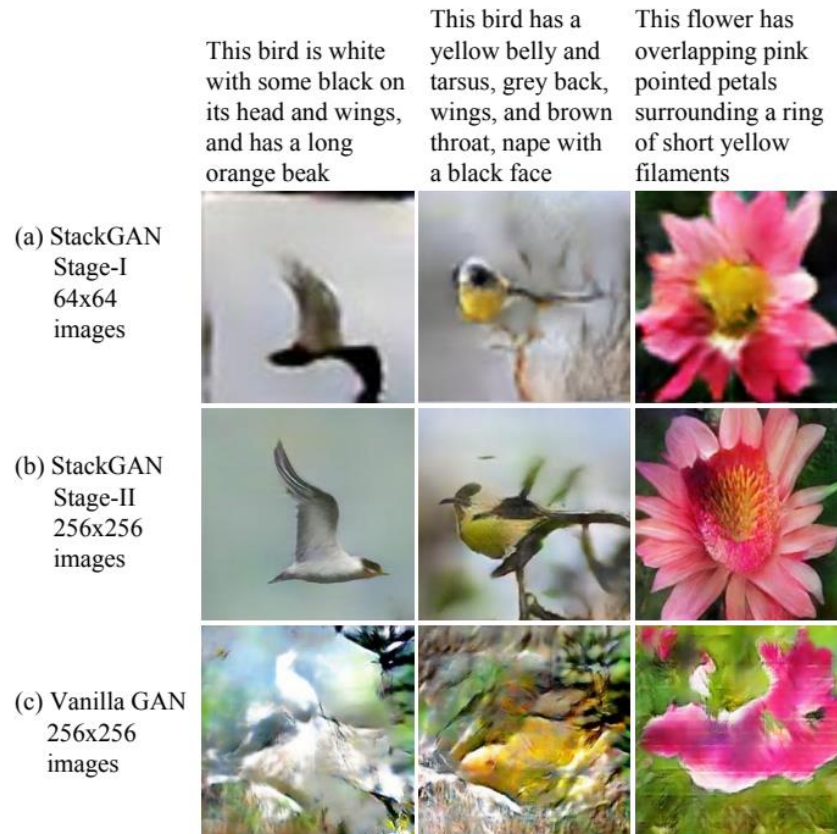


Рисунок 10.11 – Приклад роботи змагальної мережі, що породжує, для генерації фото-реалістичних зображень StackGAN у порівнянні зі звичайною GAN.

StackGAN (рисунок 10.11 – Text до Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks) – породжувальна змагальна мережа для генерації фотореалістичних зображень (256 на 256) виходячи з текстового опису. Генерувати фотореалістичні зображення на звичайних GAN складно, тому була придумана двоетапна модель генерації. Stage-I GAN малює скетчі з примітивними формами та кольорами, засновані на текстовому описі, у низькій роздільній здатності. Stage-II GAN приймає на вхід зображення з першого етапу та текстовий опис та генерує зображення у високій роздільній здатності з фото-реалістичними деталями. Щоб покращити різноманітність синтезованих зображень та стабілізувати навчання, замість CGAN використовувався метод Conditioning Augmentation [100].

Раніше використовувалися CGAN, оскільки на вхід їм можна було подавати умови, але просто додаючи шари, що збільшують розмір

зображення, досягти хороших результатів не вдалося. Тому основним завданням було підвищити роздільну здатність зображень.

Однією з ключових особливостей StackGAN є *Conditioning Augmentation*, оскільки воно дозволило розширити кількість прикладів тренувального сету шляхом невеликих випадкових змін у вихідних зображеннях, що збільшувало різноманітність даних. Як показано на рисунку, текстовий опис t кодувальником перекладається у векторне уявлення t . Раннє векторне уявлення нелінійно трансформувалося, щоб отримати приховані умовні змінні, які подавалися на вхід генератору, проте простір значень прихованих змінних має велику розмірність, що призводило до розривів у різноманітності даних, що не вигідно для генератора. Щоб позбутися цього саме потрібно Conditioning Augmentation, яке на відміну від надання фіксованих значень змінних вибирає їх з нормального розподілу $N(\mu(\varphi_t), \Sigma(\varphi_t))$, де середнє значення $\mu(\varphi_t)$ і коваріація $\Sigma(\varphi_t)$ це функції від вхідного вектора φ_t . На додаток до вже згаданого, щоб зробити різноманіття гладким і не перенавчитися, потрібно додати регуляризацию, $D_{KL}(N(\mu(\varphi_t), \Sigma(\varphi_t)) || N(0, I))$ (KL divergence) (рисунк 10.12).

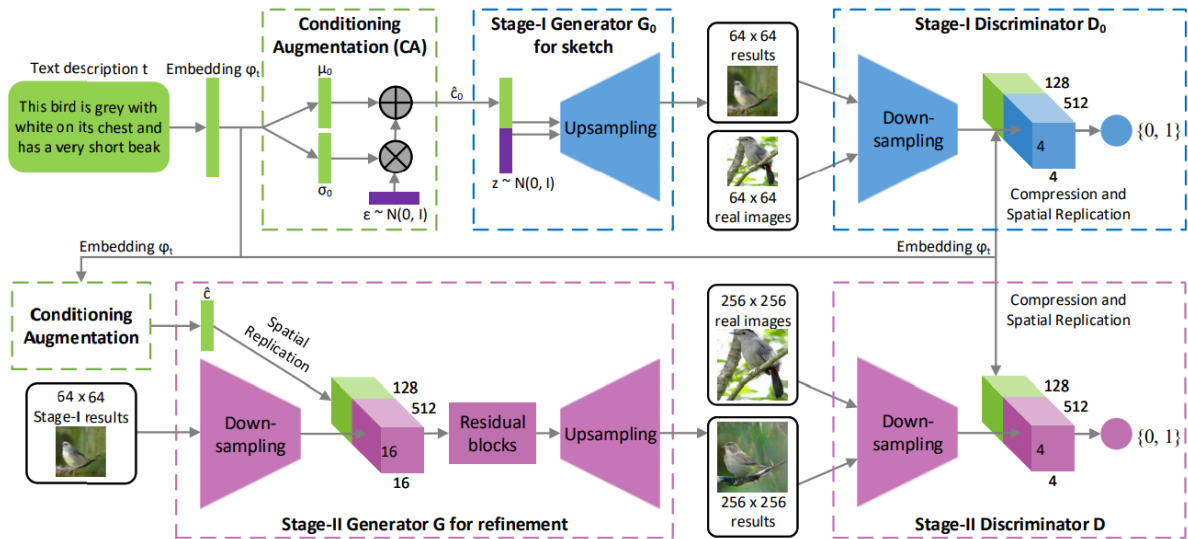


Рисунок 10.12 – Процес навчання StackGAN
(<https://arxiv.org/pdf/1612.03242.pdf>)

Stage-I GAN тренує дискримінатор D_0 і генератор G_0 почерговій максимізації LD_0 і мінімізації LG_0 , як зазначено в рівняннях:

$$L_D = E_{(I,t) \sim p_{data}} [\log D(I, \varphi_t)] + E_{s_0 \sim p_{G_0}, t \sim p_{data}} [\log(1 - D(G(s_0, \hat{c}), \varphi_t))]$$

$$L_G = E_{s_0 \sim p_{G_0}, t \sim p_{data}} [\log(1 - D(G(s_0, \hat{c}), \varphi_t))] + \lambda D_{KL}(N(\mu(\varphi_t), \Sigma(\varphi_t)) || N(0, I))$$

де реальне зображення I_0 та опис тексту t беруться з реального розподілу даних **pdata**. z – шумовий вектор взятого випадково з нормального розподілу, параметр регуляризації λ .

У зображеннях з низькою роздільною здатністю, згенерованих Stage-I GAN, зазвичай бракує яскравих деталей і можуть бути викривлення форм, деякі деталі зображення також можуть бути опущені на першому етапі. Stage-II GAN побудований над Stage-I GAN і приймає на вхід його вихід, та текстовий опис, щоб виправити та доповнити зображення. Його дискримінатор та генератор тренуються шляхом почергової максимізації LD та мінімізації LG, як показано в рівняннях:

$$L_D = E_{(I,t) \sim p_{data}} [\log D(I, \varphi_t)] + E_{s_0 \sim p_{G_0}, t \sim p_{data}} [\log(1 - D(G(s_0, \hat{c}), \varphi_t))]$$

$$L_G = E_{s_0 \sim p_{G_0}, t \sim p_{data}} [\log(1 - D(G(s_0, \hat{c}), \varphi_t))] + \lambda D_{KL}(\mathcal{N}(\mu(\varphi_t), \Sigma(\varphi_t)) || \mathcal{N}(0, I))$$

де $s_0 = G_0(z, c_0)$ результат роботи генератора Stage-I GAN та прихований параметр c подаються на вхід дискримінатору та генератору Stage-II GAN, при цьому на вхід не подається випадкове значення, як на першій стадії, оскільки вистачає подачі випадкового z на вхід Stage-I GAN. При цьому Stage-I GAN і Stage-II GAN мають різні повнорозмірні шари, щоб відрізнитися за середнім значенням і стандартним відхиленням, таким чином на різних етапах фокусуємося на різних деталях вихідного тексту.

LAPGAN (Laplacian Pyramid of Adversarial Networks)

LAPGAN – це генеративна параметрична модель, представлена пірамідою лапласіанів з каскадом згорткових нейронних мереж усередині, яка генерує зображення поступово від вихідного зображення з низькою роздільною здатністю до зображення з високою. На кожному рівні піраміди навчається згорткова генеративна модель, використовуючи підхід змагальних мереж, що породжують. Така стратегія дозволяє декомпонувати завдання генерації зображень на послідовність рівнів, що спрощує її розв'язання [101].

Піраміда лапласіанів – це лінійне оборотне уявлення зображень, що складається з набору частотних смуг зображень. Нехай $d(\cdot)$ – це операція стиснення зображення розміру $j \times j$ так, що нове зображення $d(I)$ має розміри $j/2 \times j/2$, також $u(\cdot)$ – це операція розширення така, що $u(I)$ має розміри $2j \times 2j$. Тоді піраміда гаусіанів має вигляд $G(I) = [I_0, I_1, \dots, I_k]$, де $I_0 = I$ і I_k представляє собою k раз виконане застосування $d(\cdot)$. Коефіцієнти h_k на кожному рівні піраміди лапласіанів рахуються так:

$$h_k = L_k(I) = G_k(I) - u(G_{k+1}(I)) = I_k - u(I_{k+1})$$

Інтуїтивно кожен рівень захоплює структуру зображення. Кінцевий шар піраміди лапласіанів h_k це не різниця зображень, а низькочастотне уявлення, що дорівнює гаусіану $h_k = I_k$. Реконструкція по піраміді лапласіанів відбувається зворотним проходом по ній:

$$I_k = u(I_{k+1}) + h_k$$

Підхід представлений у LAPGAN працює за таким же принципом, тільки на кожному кроці замість коефіцієнтів h_k використовуються генератори $\{G_0, G_1, \dots, G_k\}$, кожен із яких захоплює розподіл коефіцієнтів h_k для реальних зображень на різних рівнях піраміди лапласіана:

$$\tilde{I}_k = u(\tilde{I}_{k+1}) + \tilde{h}_k = u(\tilde{I}_{k+1}) + G_k(z_k, u(\tilde{I}_{k+1}))$$

Процедура семплінгу для моделі LAPGAN наведена на рисунку 10.13.

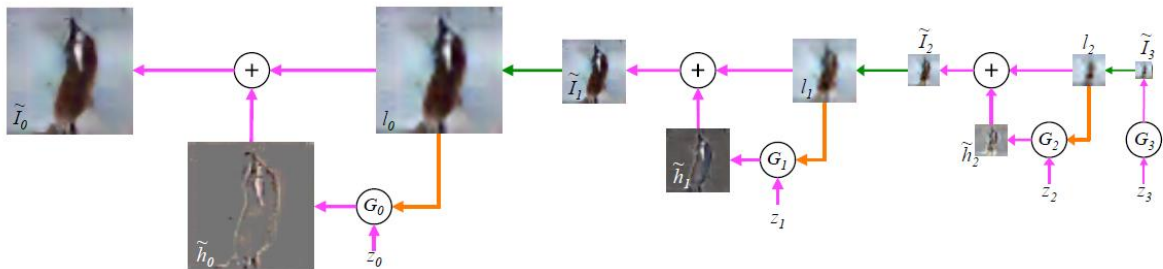


Рисунок 10.13 – Процедура семплінгу для моделі LAPGAN.

Починаємо з шуму z_3 і використовуємо генеративну модель G_3 для створення I_3 . Потім розширюємо зображення до I_2 для наступного рівня генерації G_2 . Разом із ще одним шумом z_2 отримуємо зображення відмінності I_2 . Продовжуємо процес, доки отримаємо I_0 .

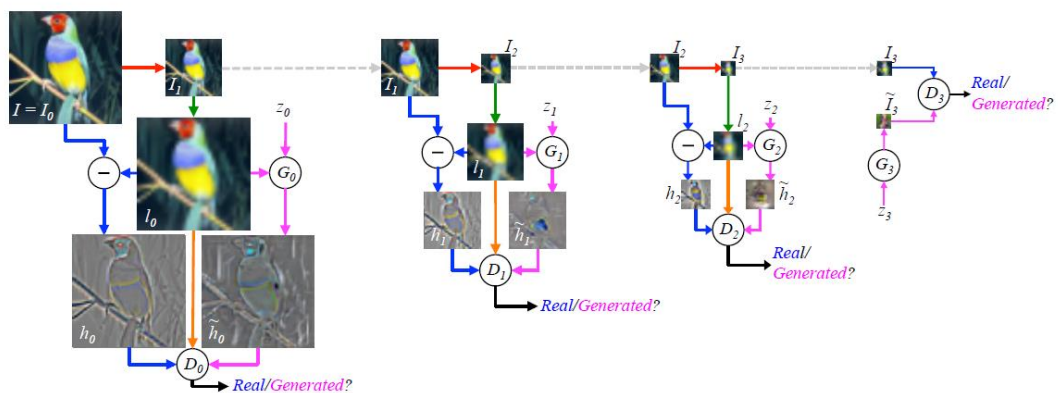


Рисунок 10.14 – Процедура навчання моделі LAPGAN.

(<https://arxiv.org/pdf/1506.05751.pdf>)

Процедура навчання LAPGAN (рисунок 10.14). Починаємо із зображення I розміру 64×64 із тренувального набору. Беремо $I_0 = I$ і стискаємо його (червона стрілка) щоб отримати I_1 ; потім розширюємо його (зелена стрілка), щоб отримати I_0 низько-частотне зображення I_0 ; з рівною ймовірністю використовуємо його для створення або реального, або згенерованого прикладу для дискримінатора D_0 . У разі реального зображення (синя стрілка) рахуємо колірний контраст $h_0 = I_0 - I_0$, який подається на вхід дискримінатору D_0 для визначення реального зображення чи ні. У разі згенерованого (рожева стрілка), генеративна мережа G_0 отримує на вхід шум z_0 та I_0 . Воно генерує колірний контраст $h_0 = G_0(z_0, I_0)$, який подається на вхід D_0 . В обох випадках дискримінатор також одержує I_0 (помаранчева стрілка). Оптимізуючи мінмакс гру умовної мережі, що породжує, G_0 вчиться генерувати реалістичну високо-частотну структуру h_0 за допомогою низько-частотного уявлення I_0 . Така процедура проходить на всіх шарах, крім останнього, де можна використовувати звичайний GAN.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Задорожний К.М. «Біологія» підручник для 8 класу з поглибленим вивченням біології закладів загальної середньої освіти (авт. Задорожний К. М., Рудич М. В.). Режим доступу: <https://uahistory.co/pidruchniki/zadorozhnij-biology-deep-level-8-class-2021/39.php>
2. Перцептрон. Матеріал з Вікіпедії. Режим доступу: <https://uk.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D1%86%D0%B5%D0%BF%D1%82%D1%80%D0%BE%D0%BD>.
3. Zhang, X. A Mathematical Model of a Neuron with Synapses based on Physiology. Nat Prec (2008). <https://doi.org/10.1038/npre.2008.1703.1>
4. Sima J. "Introduction to Neural Networks," Technical Report No. V 755, Institute of Computer Science, Academy of Sciences of the Czech Republic, 1998.
5. Kröse B., and van der Smagt P. An Introduction to Neural Networks. (8th ed.) University of Amsterdam Press, University of Amsterdam, 1996.
6. Gurney K. An Introduction to Neural Networks. (1st ed.) UCL Press, London EC4A 3DE, UK, 1997.
7. Paplinski A.P. Neural Nets. Lecture Notes, Dept. of Computer Sciences and Software Eng., Manash Universtiy, Clayton-AUSTRALIA
8. Rosenblatt, F. (1958). "The Perceptron: A Probabilistic Model For Information Storage And Organization in the Brain". Psychological Review. 65 (6): 386–408. doi:10.1037/h0042519
9. Stephan Dreiseitl, Lucila Ohno-Machado. "Logistic regression and artificial neural network classification models: a methodology review," Journal of Biomedical Informatics, 2002, pp. 352-359, [https://doi.org/10.1016/S1532-0464\(03\)00034-0](https://doi.org/10.1016/S1532-0464(03)00034-0).
10. Ripley, B. D. (1994). Neural Networks and Related Methods for Classification. Journal of the Royal Statistical Society. Series B (Methodological), 56(3), 409–456. <http://www.jstor.org/stable/2346118>

11. Haykin, Simon. Neural networks and learning machines / Simon Haykin.- 3rd ed., Pearson Education Ltd., 2008, 936p.
12. Y. LeCun et al., "Backpropagation Applied to Handwritten Zip Code Recognition," in Neural Computation, vol. 1, no. 4, pp. 541-551, Dec. 1989, doi: 10.1162/neco.1989.1.4.541.
13. J. Weng, N. Ahuja and T. S. Huang, "Cresceptron: a self-organizing neural network which grows adaptively," [Proceedings 1992] IJCNN International Joint Conference on Neural Networks, 1992, pp. 576-581 vol.1, doi: 10.1109/IJCNN.1992.287150.
14. J. J. Weng, N. Ahuja and T. S. Huang, "Learning recognition and segmentation of 3-D objects from 2-D images," 1993 (4th) International Conference on Computer Vision, 1993, pp. 121-128, doi: 10.1109/ICCV.1993.378228.
15. J. Weng, N. Ahuja and T.S. Huang. "Learning Recognition and Segmentation Using the Cresceptron," International Journal of Computer Vision 25, 109–143 (1997). <https://doi.org/10.1023/A:1007967800668>.
16. Hinton, Geoffrey E. et al. "The "wake-sleep" algorithm for unsupervised neural networks," Science, vol. 268 (5214), 1995, pp. 1158-61. doi:10.1126/science.7761831.
17. John F. Kolen; Stefan C. Kremer, "Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies," in A Field Guide to Dynamical Recurrent Networks, IEEE, 2001, pp. 237-243, doi: 10.1109/9780470544037.ch14.
18. Morgan, Nelson; Bourlard, Hervé; Renals, Steve; Cohen, Michael and Franco, Horacio. "Hybrid neural network/hidden markov model systems for continuous speech recognition," International Journal of Pattern Recognition and Artificial Intelligence, 1993, № 07 (4), pp. 899–916. doi:10.1142/s0218001493000455.
19. Rossi, Richard J. (2018). Mathematical Statistics: An Introduction to Likelihood Based Inference. New York: John Wiley & Sons. p. 227

20. Ward, Michael Don; Ahlquist, John S. (2018). Maximum Likelihood for Social Science : Strategies for Analysis. New York: Cambridge University Press
21. Press, W.H.; Flannery, B.P.; Teukolsky, S.A.; Vetterling, W.T. (1992). "Least Squares as a Maximum Likelihood Estimator". Numerical Recipes in FORTRAN: The Art of Scientific Computing (2nd ed.). Cambridge: Cambridge University Press. pp. 651–655.
22. Schwallie, Daniel P. (1985). "Positive definite maximum likelihood covariance estimators". Economics Letters. 17 (1–2): 115–117. doi:10.1016/0165-1765(85)90139-9.
23. Bottou, Léon (1998). "Online Algorithms and Stochastic Approximations". Online Learning and Neural Networks. Cambridge University Press
24. Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," JMLR, vol. 15(56), pp. 1929-1958, 2014.
25. Sergey Ioffe, Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," PMLR, vol. 37, pp. 448-456, 2015.
26. Bilmes, Jeff; Asanovic, Krste; Chin, Chee-Whye; Demmel, James (April 1997). "Using PHiPAC to speed error back-propagation learning". 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing. ICASSP. Munich, Germany: IEEE. pp. 4153-4156 vol.5. doi:10.1109/ICASSP.1997.604861
27. Hinton, Geoffrey. "Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude," p. 26, 2020.
28. Duchi, John; Hazan, Elad; Singer, Yoram (2011). "Adaptive subgradient methods for online learning and stochastic optimization," JMLR, vol. 12, pp. 2121–2159.

29. Zeiler, Matthew D. (2012). "ADADELTA: An adaptive learning rate method". arXiv:1212.5701.
30. Ruder S (2016) An overview of gradient descent optimization algorithms. arXiv. Available online at: <https://arxiv.org/pdf/1609.04747.pdf>.
31. Kingma, Diederik; Ba, Jimmy (2014). "Adam: A Method for Stochastic Optimization". arXiv:1412.6980.
32. Spall, J. C. (2000). "Adaptive Stochastic Approximation by the Simultaneous Perturbation Method". IEEE Transactions on Automatic Control. 45 (10): 1839–1853. doi:10.1109/TAC.2000.880982.
33. Polyak, Boris T.; Juditsky, Anatoli B. (1992). "Acceleration of stochastic approximation by averaging" (PDF). SIAM J. Control Optim. 30 (4): 838–855. doi:10.1137/0330046
34. Yoshua Bengio. "Practical Recommendations for Gradient-Based Training of Deep Architectures." Neural Networks: Tricks of the Trade (2012). doi:10.1007/978-3-642-35289-8_26.
35. Domínguez A. "A History of the Convolution Operation," IEEE Pulse. — 2015. — Vol. 6, no. 1. — P. 38—49
36. Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.
37. Yamashita, R., Nishio, M., Do, R.K.G. et al. Convolutional neural networks: an overview and application in radiology. Insights Imaging 9, 611–629 (2018). <https://doi.org/10.1007/s13244-018-0639-9>
38. LeCun Y, Bengio Y, Hinton G. Deep learning. Nature. 2015 May 28;521(7553):436-44. doi: 10.1038/nature14539. PMID: 26017442.
39. Yasaka K, Akai H, Abe O, Kiryu S (2018) Deep learning with convolutional neural network for differentiation of liver masses at dynamic contrast-enhanced CT: a preliminary study. Radiology 286:887–896
40. Hubel DH, Wiesel TN (1968) Receptive fields and functional architecture of monkey striate cortex. J Physiol 195:215–243

41. Dan C. Cireşan, Alessandro Giusti, Luca M. Gambardella, and Jürgen Schmidhuber. “Deep neural networks segment neuronal membranes in electron microscopy images,” In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'12), 2012. Curran Associates Inc., Red Hook, NY, USA, 2843–2851. doi: <https://dl.acm.org/doi/10.5555/2999325.2999452>.
42. A. Krizhevsky, I. Sutskever, and G. E. Hinton. “ImageNet classification with deep convolutional neural networks,” Commun. ACM, vol. 60, 6 (June 2017), 84–90. <https://doi.org/10.1145/3065386>.
43. Cireşan, D.C., Giusti, A., Gambardella, L.M., Schmidhuber, J. “Mitosis Detection in Breast Cancer Histology Images with Deep Neural Networks,” In: Mori, K., Sakuma, I., Sato, Y., Barillot, C., Navab, N. (eds) Medical Image Computing and Computer-Assisted Intervention – MICCAI 2013. MICCAI 2013. Lecture Notes in Computer Science, vol 8150. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-40763-5_51.
44. Hof, Robert D. “Is Artificial Intelligence Finally Coming into Its Own?” MIT Technology Review, MIT Technology Review, 29 Mar. 2016. [Режим доступа]: www.technologyreview.com/s/513696/deep-learning/.
45. Ciresan, Dan C. et al. “Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition.” ArXiv abs/1003.0358, 2010. [Режим доступа]: <https://arxiv.org/pdf/1003.0358.pdf>
46. Sermanet, Pierre et al. “OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks.” CoRR abs/1312.6229, 2014. [Режим доступа]: <https://arxiv.org/pdf/1312.6229.pdf>
47. Simonyan, Karen and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition.” CoRR abs/1409.1556, 2015. [Режим доступа]: <https://arxiv.org/abs/1409.1556>
48. Lin, Min et al. “Network In Network.” CoRR abs/1312.4400, 2014. [Режим доступа]: <https://arxiv.org/abs/1312.4400>

49. Szegedy, Christian et al. "Going deeper with convolutions." 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2015): 1-9. [Режим доступа]: <https://arxiv.org/abs/1409.4842>
50. Szegedy, Christian et al. "Rethinking the Inception Architecture for Computer Vision." 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 2818-2826. doi:10.1109/CVPR.2016.308
51. He, Kaiming et al. "Deep Residual Learning for Image Recognition." 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016): 770-778. doi:10.1109/cvpr.2016.90
52. P. Sermanet and Y. LeCun, "Traffic sign recognition with multi-scale Convolutional Networks," The 2011 International Joint Conference on Neural Networks, 2011, pp. 2809-2813, doi: 10.1109/IJCNN.2011.6033589.
53. Szegedy, Christian et al. "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning." AAAI (2017). doi:10.1609/aaai.v31i1.11231
54. Iandola, Forrest N. et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size." ArXiv abs/1602.07360, 2016. [Режим доступа]: <https://arxiv.org/abs/1602.07360>
55. Paszke, Adam et al. "ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation." ArXiv abs/1606.02147, 2016. [Режим доступа]: <https://arxiv.org/abs/1606.02147>
56. Chollet, François. "Xception: Deep Learning with Depthwise Separable Convolutions," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2017): 1800-1807. doi: DOI:10.1109/CVPR.2017.195
57. Howard, Andrew G. et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." ArXiv abs/1704.04861, 2017. [Режим доступа]: <https://arxiv.org/abs/1704.04861>

58. Larsson, Gustav et al. "FractalNet: Ultra-Deep Neural Networks without Residuals." ArXiv abs/1605.07648, 2017. [Режим доступа]: <https://arxiv.org/abs/1605.07648>
59. Ronneberger, Olaf et al. "U-Net: Convolutional Networks for Biomedical Image Segmentation." ArXiv abs/1505.04597, 2015. [Режим доступа]: <https://arxiv.org/abs/1505.04597>
60. Wan, Alvin et al. "NBDT: Neural-Backed Decision Trees." ArXiv abs/2004.00221, 2020. [Режим доступа]: <https://arxiv.org/abs/2004.00221>
61. T. Robinson, "A real-time recurrent error propagation network word recognition system," [Proceedings] ICASSP-92: 1992 IEEE International Conference on Acoustics, Speech, and Signal Processing, 1992, pp. 617-620, vol.1, doi: 10.1109/ICASSP.1992.225833.
62. A. Waibel, T. Hanazawa, G. Hinton, K. Shikano and K. J. Lang, "Phoneme recognition using time-delay neural networks," in IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 37, no. 3, pp. 328-339, March 1989, doi: 10.1109/29.21701.
63. J. Baker, Li Deng, Jim Glass, S. Khudanpur, C.-H. Lee, N. Morgan, D. O'Shaughnessy. "Research Developments and Directions in Speech Recognition and Understanding, Part 1" IEEE Signal Processing Magazine. № 26 (3), pp. 75–80. doi:10.1109/msp.2009.932166.
64. Y. Bengio. Artificial Neural Networks and their Application to Speech / Sequence Recognition, McGill University Ph.D. thesis. 1991. [Режим доступа]: https://escholarship.mcgill.ca/concern/file_sets/kw52j887f?locale=en
65. L. Deng, K. Hassanein, M. Elmasry. "Analysis of correlation structure for a neural predictive model with applications to speech recognition," Neural Networks, vol. 7 (2), 1994, pp. 331–339. doi:10.1016/0893-6080(94)90027-2.
66. Machine Learning I Week 14: Sequence Learning Introduction, Alex Graves, Technische Universitaet Muenchen. [Режим доступа]:

http://archive.www6.in.tum.de/www6/pub/Main/TeachingWs2009MachineLearning/Slides_09.pdf

67. CSC2535 2013: Advanced Machine Learning, Lecture 10: Recurrent neural networks, G. Hinton, University of Toronto. [Режим доступа]: <https://www.cs.toronto.edu/~hinton/csc2535/notes/lec10new.pdf>
68. CS224d Deep NLP, Lecture 8: Recurrent Neural Networks, Richard Socher, Stanford University. [Режим доступа]: <https://cs224d.stanford.edu/lectures/CS224d-Lecture8.pdf>
69. Supervised Sequence Labelling with Recurrent Neural Networks, Alex Graves, Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigten Dissertation. [Режим доступа]: <https://www.cs.toronto.edu/~graves/phd.pdf>
70. The Unreasonable Effectiveness of Recurrent Neural Networks, Andrej Karpathy, blog About Hacker's guide to Neural Networks. [Режим доступа]: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>
71. Understanding LSTM Networks, Christopher Olah [Режим доступа]: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>
72. Ryan Kiros, Yukun Zhu, Ruslan Salakhutdinov, Richard S. Zemel, Antonio Torralba, Raquel Urtasun, and Sanja Fidler, “Skip-thought vectors,” In Proceedings of the 28th International Conference on Neural Information Processing Systems, Vol. 2 (NIPS'15), MIT Press, Cambridge, MA, USA, 3294–3302. <https://dl.acm.org/doi/10.5555/2969442.2969607>
73. Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. “Language modeling with gated convolutional networks,” In Proceedings of the 34th International Conference on Machine Learning, vol. 70 (ICML'17), JMLR.org, pp. 933–941.
<https://dl.acm.org/doi/10.5555/3305381.3305478>
74. Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Rezende, Daan Wierstra. “DRAW: A recurrent neural network for image generation,” Proceedings of the 32nd International Conference on Machine Learning,

- PMLR 37:1462-1471, 2015. [Режим доступа]: <https://proceedings.mlr.press/v37/gregor15.html>.
75. Hang Chu, Raquel Urtasun, and Sanja Fidler. “Song From PI: A Musically Plausible Network for Pop Music Generation,” arXiv preprint arXiv:1611.03477, 2016. [Режим доступа]: <https://arxiv.org/pdf/1611.03477.pdf>
 76. Zheng, Shuai, et al. “Conditional random fields as recurrent neural networks,” Proceedings of the IEEE International Conference on Computer Vision. 2015, pp. 1529-1537. doi:10.1109/ICCV.2015.179.
 77. Ramanathan, Vignesh, et al. “Detecting events and key actors in multi-person videos,” arXiv preprint arXiv:1511.02917, 2015. [Режим доступа]: <https://arxiv.org/abs/1511.02917>
 78. Shah, Rajiv, and Rob Romijnders. “Applying Deep Learning to Basketball Trajectories,” arXiv preprint arXiv:1608.03793, 2016. [Режим доступа]: <https://arxiv.org/pdf/1608.03793.pdf>
 79. Moez Baccouche, Franck Mamalet, Christian Wolf, Christophe Garcia, Atila Baskurt. “Action Classification in Soccer Videos with Long Short-Term Memory Recurrent Neural Networks,” 20th International Conference on Artificial Neural Networks (ICANN), Sep 2010, Thessaloniki, Greece. pp.154-159, doi:10.1007/978-3-642-15822-3_20.
 80. Chung, Junyoung, et al. “Empirical evaluation of gated recurrent neural networks on sequence modeling,” arXiv preprint arXiv:1412.3555 (2014). [Режим доступа]: <https://arxiv.org/abs/1412.3555>
 81. Greff, Klaus et al. “LSTM: A Search Space Odyssey,” IEEE Transactions on Neural Networks and Learning Systems, vol. 28, 2017, pp. 2222-2232 doi:10.1109/tnnls.2016.2582924.
 82. A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, J. Schmidhuber. “A Novel Connectionist System for Improved Unconstrained Handwriting Recognition,” IEEE Transactions on Pattern

- Analysis and Machine Intelligence, vol. 31, no. 5, 2009. doi: 10.1109/TPAMI.2008.137.
83. F. Gomez, J. Schmidhuber, R. Miikkulainen. "Accelerated Neural Evolution through Cooperatively Coevolved Synapses," Journal of Machine Learning Research (JMLR), vol. 9, pp. 937-965, 2008. doi: <https://dl.acm.org/doi/10.5555/1390681.1390712>.
 84. J. Schmidhuber, D. Wierstra, M. Gagliolo, F. Gomez. "Training Recurrent Networks by Evolino," Neural Computation, vol. 19(3), pp. 757-779, 2007. doi: <https://doi.org/10.1162/neco.2007.19.3.757>
 85. F. Gers, N. Schraudolph, J. Schmidhuber. "Learning precise timing with LSTM recurrent networks," Journal of Machine Learning Research, vol. 3, pp. 115-143, 2002. doi: <https://doi.org/10.1162/153244303768966139>.
 86. J. Schmidhuber, F. Gers, D. Eck. J. Schmidhuber, F. Gers, D. Eck. "Learning nonregular languages: A comparison of simple recurrent networks and LSTM," Neural Computation, vol. 14(9), pp. 2039-2041, 2002. doi: <https://doi.org/10.1162/089976602320263980>.
 87. F. A. Gers and E. Schmidhuber, "LSTM recurrent networks learn simple context-free and context-sensitive languages," in IEEE Transactions on Neural Networks, vol. 12, no. 6, pp. 1333-1340, Nov. 2001, doi: 10.1109/72.963769.
 88. F. A. Gers and J. Schmidhuber and F. Cummins. "Learning to Forget: Continual Prediction with LSTM," Neural Computation, vol. 12(10), pp. 2451-2471, 2000, doi: 10.1049/cp:19991218.
 89. S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory," Neural Computation, vol. 9(8), pp. 1735-1780, 1997. doi: <https://doi.org/10.1162/neco.1997.9.8.1735>
 90. Natekin A and Knoll A. "Gradient boosting machines, a tutorial," Front. Neurorobot. Vol. 7:21. 2013. doi: 10.3389/fnbot.2013.00021

91. Cho, Kyunghyun et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation." EMNLP, 2014. doi:10.3115/v1/D14-1179
92. Yao, Kaisheng et al. "Depth-Gated LSTM." ArXiv abs/1508.03790, 2015. [Режим доступа]: <https://arxiv.org/abs/1508.03790>
93. Koutník, Jan et al. "A Clockwork RNN." ICML, 2014. [Режим доступа]: <https://arxiv.org/abs/1402.3511?context=cs.NE>
94. Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. 2015. An empirical exploration of recurrent network architectures. In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15). JMLR.org, 2342–2350. doi: <https://dl.acm.org/doi/10.5555/3045118.3045367>
95. Mao, Junhua et al. "Deep Captioning with Multimodal Recurrent Neural Networks (m-RNN)." arXiv: Computer Vision and Pattern Recognition, 2015. [Режим доступа]: <https://arxiv.org/abs/1412.6632>
96. J. Langr and V. Bok. GAN in Action: Deep Learning with Generative Adversarial Networks, Manning, 2019, 276p.
97. Ramponi, Giorgia et al. "T-CGAN: Conditional Generative Adversarial Network for Data Augmentation in Noisy Time Series with Irregular Sampling." ArXiv abs/1811.08295, 2018. [Режим доступа]: <https://arxiv.org/abs/1811.08295>
98. Mirza, Mehdi and Simon Osindero. "Conditional Generative Adversarial Nets." ArXiv abs/1411.1784, 2014. [Режим доступа]: <https://arxiv.org/abs/1411.1784>
99. Radford, Alec et al. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks." CoRR abs/1511.06434, 2016. [Режим доступа]: <https://arxiv.org/abs/1511.06434>.
100. H. Zhang et al., "StackGAN: Text to Photo-Realistic Image Synthesis with Stacked Generative Adversarial Networks," 2017 IEEE International

- Conference on Computer Vision (ICCV), 2017, pp. 5908-5916, doi: 10.1109/ICCV.2017.629.
101. Emily Denton, Soumith Chintala, Arthur Szlam, and Rob Fergus. 2015. Deep generative image models using a Laplacian pyramid of adversarial networks. In Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'15). MIT Press, Cambridge, MA, USA, 1486–1494. doi: <https://dl.acm.org/doi/10.5555/2969239.2969405>
 102. [Электронный ресурс] Tutorials on Tensorflow. [Режим доступа]: <https://www.tensorflow.org/tutorials/>
 103. [Электронный ресурс] One-vs-All Classification Using Logistic Regression. [Режим доступа]: <https://utkuufuk.com/2018/06/03/one-vs-all-classification/>
 104. [Электронный ресурс] Machine Learning Co ban-Bài 10: Logistic Regression. [Режим доступа]: <https://machinelearningcoban.com/2017/01/27/logisticregression/>
 105. [Электронный ресурс] Common Loss functions in machine learning. [Режим доступа]: <https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23>
 106. [Электронный ресурс] Prince Grover, 5 Regression Loss Functions All Machine Learners Should Know. [Режим доступа]: <https://heartbeat.comet.ml/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0>
 107. [Электронный ресурс] Difference Between Classification and Regression in Machine Learning. [Режим доступа]: <https://machinelearningmastery.com/classification-versus-regression-in-machine-learning/>
 108. [Электронный ресурс] Sowmya Yellapragada, Common loss functions that you should know! [Режим доступа]:

- <https://medium.com/ml-cheat-sheet/winning-at-loss-functions-common-loss-functions-that-you-should-know-a72c1802ecb4>
109. [Электронный ресурс] ML | Common Loss Functions. [Режим доступа]: <https://www.geeksforgeeks.org/ml-common-loss-functions/>
 110. [Электронный ресурс] Jason Brownlee, How to Choose Loss Functions When Training Deep Learning Neural Networks. [Режим доступа]: <https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/>
 111. [Электронный ресурс] Phuc Truong, Loss functions: Why, what, where or when? [Режим доступа]: <https://medium.com/@phuctrt/loss-functions-why-what-where-or-when-189815343d3f>
 112. [Электронный ресурс] Rafay Khan, Nothing but NumPy: Understanding & Creating Binary Classification Neural Networks with Computational Graphs from Scratch. [Режим доступа]: <https://towardsdatascience.com/nothing-but-numpy-understanding-creating-binary-classification-neural-networks-with-e746423c8d5c>
 113. [Электронный ресурс] Kamil Mysiak, Binary Classification Example. [Режим доступа]: <https://towardsdatascience.com/binary-classification-example-4190fcfe4a3c>
 114. [Электронный ресурс] CS231n: Deep Learning for Computer Vision [Режим доступа]: <http://cs231n.stanford.edu/syllabus.html>
 115. [Электронный ресурс] Differences between L1 and L2 Loss Function and Regularization. [Режим доступа]: <http://www.chioka.in/differences-between-l1-and-l2-as-loss-function-and-regularization/>
 116. [Электронный ресурс] Empirical Risk Minimization: Cornell. [Режим доступа]: <https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote10.html>

117. [Электронный ресурс] Steve Dye, Quantile Regression. [Режим доступа]: <https://towardsdatascience.com/quantile-regression-ff2343c4a03>
118. [Электронный ресурс] Benoit Descamps. “Regression prediction intervals using xgboost”. [Режим доступа]: <https://towardsdatascience.com/regression-prediction-intervals-with-xgboost-428e0a018b>
119. [Электронный ресурс] Rodriguez, Robert N.. “Five Things You Should Know about Quantile Regression.”, 2017. [Режим доступа]: <https://www.semanticscholar.org/paper/Five-Things-You-Should-Know-about-Quantile-Rodriguez/7e824688e34f6010ee203c86643b17ccbd9acf90>

РЕЄСТР. № НП 22/23-133. Обсяг 10,0 авт. арк.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ПРОСПЕКТ ПЕРЕМОГИ, 37, м. Київ, 03056

[HTTPS://KPI.UA](https://kpi.ua)

**СВІДОЦТВО ПРО ВНЕСЕННЯ ДО ДЕРЖАВНОГО РЕЄСТРУ ВИДАВЦІВ, ВИГОТОВЛЮВАЧІВ
І РОЗПОВСЮДЖУВАЧІВ ВИДАВНИЧОЇ ПРОДУКЦІЇ ДК № 5354 від 25.05.2017 р.**